Model Based Testing distributed Identity and Access Management software: Verifying OIDCfed

Jouke M. Roorda
jouke.roorda@student.uva.nl

August 2019, 39 pages

University supervisor	Ana Oprescu
Host organization	Axini
Host supervisor	Machiel van der Bijl

Universiteit van Amsterdam
Faculteit der Natuurwetenschappen, Wiskunde en Informatica
Master Software Engineering
http://www.software-engineering-amsterdam.nl

Abstract

In the field of Federated Identity and Access Management (FIAM), trust is everything, as the identities a system provides act as the basis for granting access. If an untrusted system can provide identities, it might indirectly provide administrator access to the wrong people, or leak personal data.

To see if the newly developed OpenID Conenct Federations (OIDCfed) reference implementation, which by its nature makes decisions about trusting certain third parties (i.e. handles federated trust), conforms with its corresponding specification, this research entails the exploration of using Model Based Testing (MBT) in the field of FIAM. Model Based Testing would seem applicable as there is a written, detailed specification - which can be turned into a model - and a leading implementation - which can be tested by the tests that are developed based on the model. This MBT approach is in addition to the unit tests that have already been developed as part of the reference implementation.

The systematic MBT approach lead to the finding of non security related issues in the source code and the configuration of the signing service, where experienced testers did not, meaning that the reference implementation and configuration did not fully comply with the specification. The issues that have been found have either been fixed in the public repository, or have been passed on to the editor and developer of the OIDCfed work.

Contents

Chapter 1

Introduction

Nikhef, the Dutch National Institute for Sub-atomic Physics, takes part in EU collaborations regarding authentication and authorization infrastructure (AAI) for distributed infrastructures for research – so called e-Infrastructures. It provides both operational services such as trust anchor distributions, as well as some toolkits for AAI components that are used in both European and global research infrastructures, such as the WLCG[1] computing grid for the analysis of LHC high-energy physics data, and EGI[2] and EOSC-HUB[3] generic e-Infrastructures. Most of these infrastructures are either moving to, or considering to move to OpenID Connect(OIDC, a platform for federated login) as a possible identity service and are – among others, for scalability reasons – in need of a federated approach. One of the approaches to do this has been under development as the official OpenID standard for Federations. Lead by Roland Hedberg, with input from the European Research community, this standard has been developed together with a reference implementation. In order to understand if the software for OpenID Connect Federations can be seen as trusted, there is a need for validation.

1.1 Problem statement

Open collaborations have the potential to incorporate people with different opinions and ideas, and thus the specification and reference implementation have - so far - been the subject of at least one significant overhaul. Changing the specification, and in turn interpreting it and changing the reference implementation, can lead to inconsistencies and skewed implementations. Not following the collectively written specification can lead to issues regarding trust, privacy and information security. With developers having the tendency to copy code, or potentially preferring to look at an implementation, rather than developing a library from the written specification, there is a strong need for a robust reference implementation.

At the same time, Axini, where this research has been hosted, is interested in the various fields where Model Based Testing can be applied - them being a link between the fairly academic MBT and industry. Security is of course, by it's nature, a field where everything has to do exactly what it's supposed to. The result of this research brings the application of MBT to AAI closer to the non-academic world.

1.1.1 Research questions

To ensure the reference implementation is accurate and consistently follows the specification, we will explore the usage of Model Based Testing (MBT) to investigate the implementation's compliance. Model Based Testing can prevent Nikhef and other organizations from getting into potential operational trouble or issues regarding interoperable trust relationships, by enabling them to use verified software. In this research, we aim to answer the following questions:

RQ1. Is it possible to create a model from the written specification that can be used to check the reference implementation?

RQ2. Does the usage of MBT result in the finding of errors in the reference implementation?

RQ3. To what extent are the errors that are found by MBT violations of the specification?

RQ4. How does the usage of MBT compare to manual testing in terms of incorrect behavior detection?

[1] http://wlcg.web.cern.ch

[2] https://www.egi.eu

[3] https://www.eosc-hub.eu

1.1.2 Research method

Axini, an Amsterdam based company, has developed a platform that applies academic principles of Model Based Testing to commercial software. Consuming models, their platform generates tests that prove or disprove so-called input-output compliance (ioco) of the test subject. During this research a model for the OpenID Federation specification will be developed and ingested by Axini's platform to run against a test setup, using example services that are included in the reference implementation.

In extension to the reference implementation, any other implementation of the same specification can be checked with the tools that are to be created, as they should yield the same input/output behaviour as the reference implementation. This means that all the results in this research are expected to be reproducible, given the right environment. With OpenID Federations being used over https, it could even be that the tests are technology agnostic. This is, however, not a primary goal for this research.

During the course of this research, the specification has changed with the implementation not always keeping up. The focus of this research has therefore been set to draft six of the specification. This version has, at the time of writing, been the latest version with a full implementation next to it. The OIDCfed editor has to me expressed his wish to get to a release candidate with the written text, before catching up with the implementation - which will most likely be late September, outside the timeline of this research.

As part of the European Community, it is our intention to give any findings back to the community, either by fixing problems or by discussing findings.

1.2 Contributions

This research makes the following contributions:

1. Verifying or disproving the compliance of the reference implementation with regards to the specification
2. By modeling the specification, the specification will be interpreted in a systematic way, potentially highlighting inconsistencies and other issues
3. Comparing the usage of MBT to manual testing when it comes to developing implementations of a specification
4. Exploring the usage of MBT in the field of security software
5. The improvement of OIDCfed where possible

Chapter 2

Background

2.1 Authentication protocols

OpenID Connect (OIDC)[1], like SAML[2], is an authentication and authorization platform. Where SAML has found its way into research communities with international initiatives like eduGAIN[1] and national uses such as DigiD[2], OIDC - which was introduced almost 10 years later[3] - found support from so-called Social ID providers such as Google[4] and PayPal[5].

All these platforms suit the purpose of centralizing login systems in their own way, meaning they offer to connect one or many services to one or many sources of identity. While there is research happening on so-called non-web access (in the web browser sense), the main focus of these protocols, and this research, is web access. This means centralizing the way that users log in to web sites that belong to a certain community or group.

2.2 Software Testing

When building a software system, it is desirable to know if this system behaves as intended, especially when it is to be used by someone other than the developer. Already in the 1970's, Royce - in his well known paper on software development processes - writes the following: "Test every logic path in the computer program at least once with some kind of numerical check. If I were a customer, I would not accept delivery until this procedure was completed and certified. This step will uncover the majority of coding errors." [3]

These tests can be done by a person, or by a programmed system. Both approaches have various methods and technologies related to them. In the latter category, 'unit testing' is of particular interest, as it has been used by the developer of the System Under Test (SUT) of this research. Unit testing is a practice in which a piece of the system is tested by invoking it under certain circumstances, and checking for expected changes in the environment (e.g. a resulting value, a state change, or the invocation of external behaviour). These tests can be very fine grained, and can be very much embedded into the system itself. Unit tests consist of test cases that are usually developed by either the software developer or a software tester. With the rise of Test Driven Development and Continuous Integration (coupled with automatic testing), testing has come to play a significant role in the life cycle of software.

Software Testing can happen on various levels with various approaches. Take the following image.

[1] https://technical.edugain.org
[2] https://www.digid.nl
[3] SAML2.0 was introduced in 2005, while OIDC was introduced in 2014
[4] https://developers.google.com/identity/protocols/OpenIDConnect
[5] https://developer.paypal.com/docs/integration/direct/identity/log-in-with-paypal/

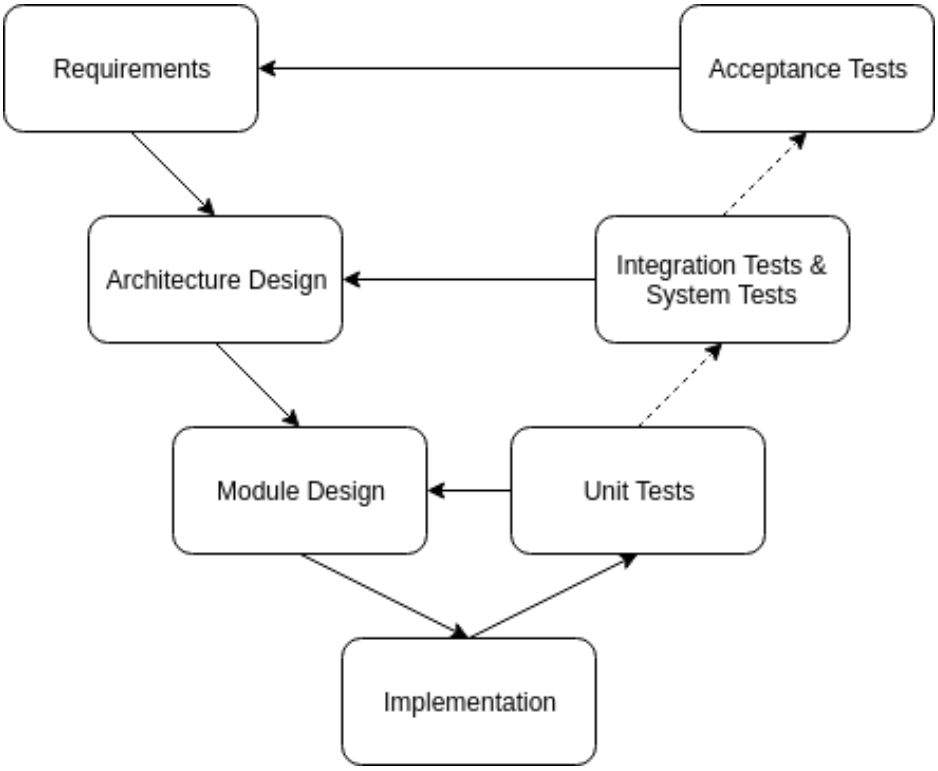


Figure 1 *Software Testing V-model*

This image shows how the various levels of testing relate to the various design phases that are involved in software development. Unit tests will test very fine grained parts of the software system, such as 'functions' or 'routines', while integration tests will verify components, such as (part of) an API. Lastly acceptance tests will verify if the software does what the client wants it to do. These levels of testing come with different challenges and solutions - and might be done by different stake holders in the process. They also presume different knowledge about the system. Where unit tests can be very fine grained and even influence the environment, or encapsulate 'hidden' parts of software to see how they were used - this is called white box testing, higher level tests will most likely not be interested in the inner workings of the system, only caring about what happens in the long run - so called black box testing. Black box testing entails the assumption that when the right things happen, they also happened for the right reason, as it cannot 'look into' the system.

2.3 Model Based Testing

Royce's quote in the section above is immediately followed by: "...While this test procedure sounds simple, for a large, complex computer program it is relatively difficult to plow through every logic path with controlled values of input. In fact there are those who will argue that it is very nearly impossible. In spite of this I would persist in my recommendation that every logic path be subjected to at least one authentic check."

A potential counter measure to the concern that Royce is raising is Model Based Testing. Determining tests for every 'logic path' can be hard and time consuming. What should happen, and what should not happen are all cases that are to be considered when testing systems. While unit tests rely on a person thinking of how to approach the behavior of systems, MBT works in a different way: a person writes down what a system is supposed to do, while the framework then thinks of ways of verifying this behaviour. For systems that already have a specification, or a description of work at least, the step to developing a model might not have to cost a lot of effort. Should not such a specification exist, then the creation of the model is about as time-consuming as developing a formal written specification, but with the added benefit of being able to automatically generating tests. This can be seen in the image below.[4]

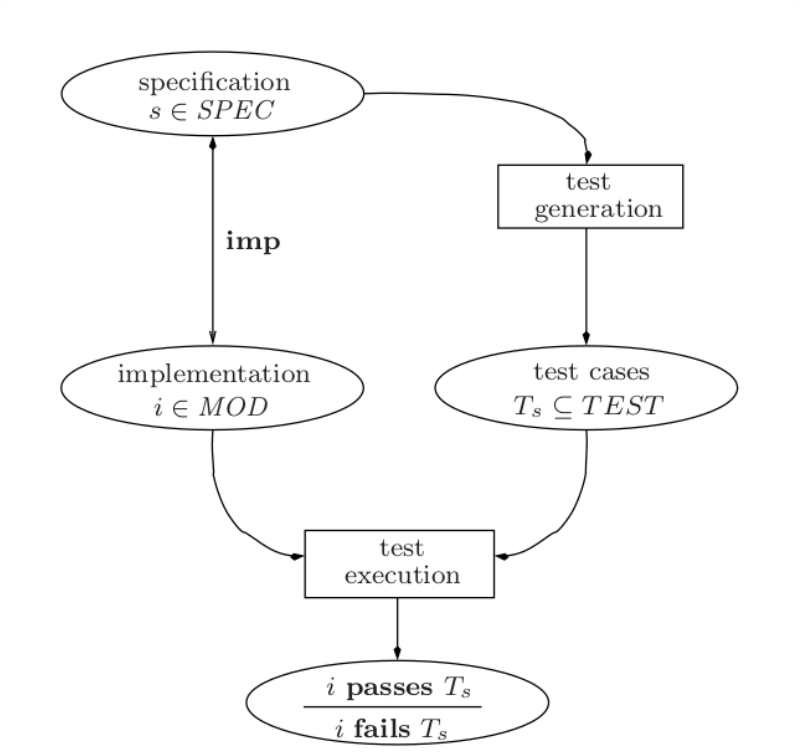


Figure 2 *Relation between specification, implementation and verification*

With the test being based on a model, the reproductability of the test results also increases, making it easier to verify corrections that have been made in the system. This stems from the systematic aspect of MBT, which indirectly also adds the potential to uncover issues with configuration and environment - which otherwise might have been sheltered by the test environment. A short overview of differences between manual, unit and model based testing can be found below, where M signifies a manual step, and A an automatic step.

	process	hand	script	MBT
Design	Make Specification	M	M	
	Make Model			M
Testing	Make Test	M	M	A
	Predict outcome	M	M	A
	Script Test		M	A
	Execute Test	M	A	A
	Check Outcome	M	A	A

Table 3 *Comparison between manual, scripted and model based testing*

The model that is defined can be seen as a Labelled Transition System, where the software is specified as a set of states, and the relation between them. These relations are labelled represent the actions that are taken to get to the next state. In the case of Model Based Testing, as it is a form of black box testing, these states are observable, as it is impossible to see if an unobservable state has been reached.

2.4 Input-Output conformance

In his work, Tretmans defines *conf* as "a system does what it is supposed to do, and does not what it is not allowed to do"[5]. This serves as the basis for *ioconf*. Undefined behaviour as a response to undefined inputs are not relevant, as long as the defined inputs result in the defined outputs.[5]

In later work by Tretmans, based on the 1996 paper, *ioco*, input-output conformance is 'informally defined' as: "an implementation i [...] is ioco-conforming to specification s [...] if any experiment derived from s and executed on i leads to an output from i that is foreseen by s."[4] This is a very practical way of

testing software systems, as it verifies observable behaviour, and thus should be implementation-agnostic. See also this formal definition[4]:

$$i \,\mathbf{ioco}\, s \;\Leftrightarrow_{\text{def}}\; \forall \sigma \in Straces(s) : out(i \,\mathbf{after}\, \sigma) \subseteq out(s \,\mathbf{after}\, \sigma)$$

Definition 4 *Formal ioco definition*

Ioco serves as the basis for the Axini platform, that provides the System Under Test with 'stimuli', and observes the responses. The system is then able to verify if the output constraints, based on the input, are observed.

2.4.1 Exhaustiveness of test sets

With most systems, and especially those that take input and output, there is an enormous number of tests possible, potentially approaching infinity. A set that contains all these tests is called *complete*. If every and all of those test cases were possible in a limited amount of time, the only - paradoxical - outcome would be that the system is insecure.

As close-to-infinite test paths are also close-to-impossible, smaller concise test sets have to be marked as sufficient. These test sets are called *sound*. Being *sound* is defined as: every failing test should indeed be non-conforming to the specification, but not per se the other way around: it does explicitly not mean that every passing test is necessarily conforming.

The extent to which non-conforming cases are actually failing during the test set is called *exhaustiveness*. If a test set is *exhaustive*, this means all non-conforming implementations are detected.[4]

2.5 OpenID Federations

With OIDC having desirable features such as simplicity of implementation for application developers, the usage of a compact JSON format and the ability to integrate with authorization frameworks such as OAuth that support non-web and mobile use cases, combined with an increasing user base, there is a growing demand from research communities to be able to use an OIDC Identity Provider as a source of authentication information. However, in contrast to most commercial services, the research community model does not match the one-to-many view that foresees a single (or few) Identity Providers connecting to many services, but rather many Identity Providers to one (the research community's) service, or even many-to-many interactions. To prevent having to connect each service with each provider manually – which includes having to exchange keys and metadata in an external, trusted manner, a specification has been introduced, simplifying access by trusted automation.

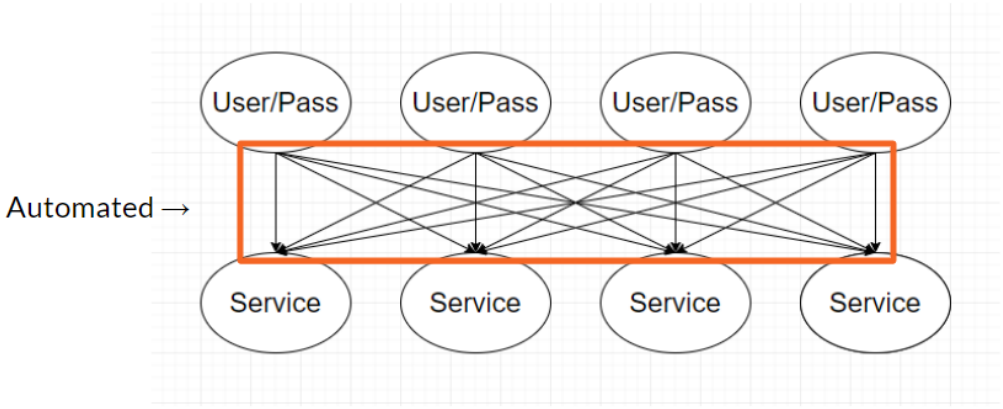


Figure 5 *OIDCfed automates trust relations, minimizing the need for manual metadata exchange*

As in a standard situation, every service has to trust every source of identity, it should be easy to imagine the amount of work that has to happen in establishing trust relations, should there be no other way to automate the trusted exchange.

Not unlike OpenID Connect, Federations consist of an OP (OpenID Provider) and an RP (Relying Party). To accommodate for the federation of trust, these have to incorporate the verification of trust chains, and the ability to select trust anchors. These trust anchors are a new component, introduced in the federation, and could be compared to, lacking a better example, a certificate authority in the context

of public key infrastructure. Examples of trust anchors could be: on a company level, e.g. Nikhef, on a national level, e.g. SURFconext or on an international (collaboration) level, e.g. eduGAIN (a GEANT service), WLCG (CERN, a worldwide NGO) or LHCb (one of the CERN experiments), to name a few. Fictional example chains can be found in appendix 2.

The addition of federated trust means the addition of communication with regards to trust. Under normal circumstances, where trust is already handled in another fashion, OpenID Connect consists of at least the following communication, as seen in the specification of the OIDC Core.[1]

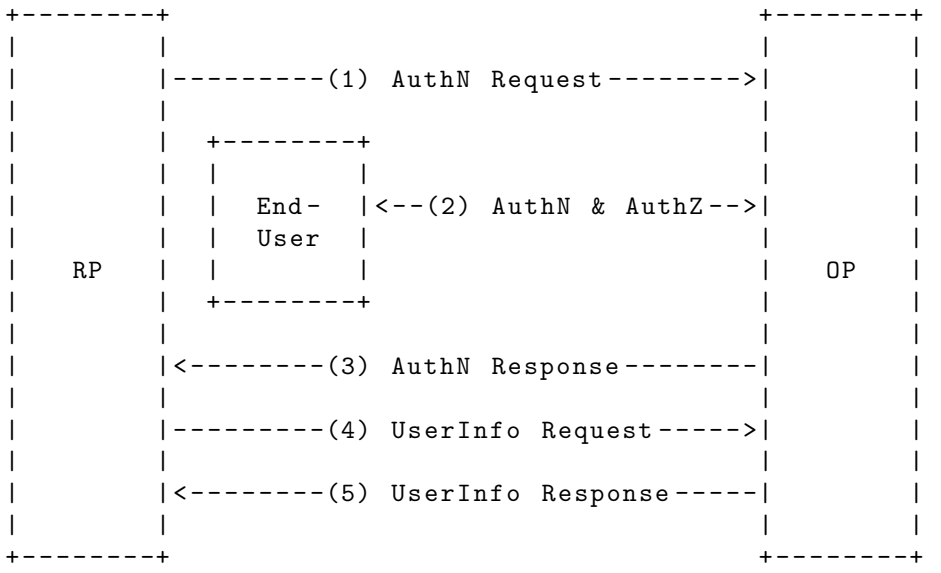


Figure 6 *Standard, non-federated, OIDC communication*

Clearly, the RP (i.e. 'Service') initiates the log in procedure, for instance because the user presses a log in button, or because there is an attempt to access a protected resource (imagine directly trying to access an order confirmation in the case of a web shop). The user then logs in on the OpenID Provider, using whatever it accepts (e.g. username and password, tokens, text messages, certificates, etc.). This logging in does not necessarily have to happen in the application that requested the user to log in - examples of this can be seen in mobile applications, where logging in can happen in the browser, and does not necessarily in the application itself. Upon completion of this log in procedure, the application can access some information about the user.

To determine if the OpenID Provider is to be trusted, signed Entity Statements have been introduced - which are handled by a new component: the Signing Service. This service can express trust in Entity Statements, and with it in components in the OIDCfed environment. Common usages are trusting OpenID Providers - marking them as legitimate providers in an attempt to prevent spoofing for instance, or marking them as part of a community - while Signing Services can also express trust in other Signing Services, in an attempt to create a collection of federations, or to express trust in a certain service's way of working. This means OIDCfed environments can consist of chains that have to be followed to verify trust in an OpenID Provider.

Take the following trust chain. There is an OpenID Provider that is trusted by Signing Service 1, which is trusted by Signing Service 2, which is trusted by Signing Service 3.

Figure 7 *Simple OIDCfed Trust Chain*

Assume there is a Relying Party that wants to log in using the OP in the above chain. Because the RP has federated its trust, it first has to verify if the OP is (still) trusted. In the configuration of the RP, any (or none, for that matter) of the Signing Services can be marked as trusted. In this example, we can assume that it trusts SignServ 2. What will happen is that the OP will be asked to issue a statement about itself. An example of such a statement is given in the appendix of the OIDCfed specification.

```
{
```

```
  "iat": 1539174048,
  "exp": 1539177648,
  "iss": "https://umu.se/openid",
  "sub": "https://umu.se/openid",
  "metadata": {
    "openid_provider": {
      "federation_types_supported": ["automatic", "explicit"],
      "authorization_endpoint":
        "https://idp.umu.se/openid/authorization",
      "token_endpoint": "https://idp.umu.se/openid/token",
      "response_types_supported": ["code", "code id_token", "token"],
      "grant_types_supported": [
        "authorization_code",
        "implicit",
        "urn:ietf:params:oauth:grant-type:jwt-bearer"
      ],
      "subject_types_supported": ["pairwise", "public"],
      "id_token_signing_alg_values_supported": ["RS256"],
      "logo_uri":
        "https://www.umu.se/img/umu-logo-left-neg-SE.svg",
      "policy_uri":
        "https://www.umu.se/en/website/legal-information/"
    }
  },
  "authority_hints": {
    "https://www.sunet.se/swamid": ["https://edugain.org/oidc"],
    "https://kalmar2.org/openid": []
  },
  "jwks": {
    "keys": [
      {
        "kty": "RSA",
        "alg": "RS256",
        "n": "z1V1kyi6qwmXfKsfhVqKUMmQH3AixN...",
        "e": "AQAB",
        "key_ops": ["verify"],
        "ext": true,
        "kid": "8S9-dy4GN8_-z...",
        "use": "sig"
      }
    ]
  }
}
```


Figure 8 *Example OpenID Provider Entity Statement*

Note that the iss(uer) and sub(ject) are the same, as the OP is handing out information about itself. Of course, this is only an example, and could in different set ups have different contents, but take note of the authority_hints. These are Signing Services that are, according to the OP, willing to express trust in the OP. To relate to the chain that has been visualized before, both sunet and kalmar2 could be "SignServ 1". The edugain url that is placed as a 'parent' of sunet, can be "SignServ 2", but kalmar2 can also provide a "SignServ 2" when asked to do so. These are all mere suggestions. The retrieval of the OP's Entity Statement is the first step in establishing trust.

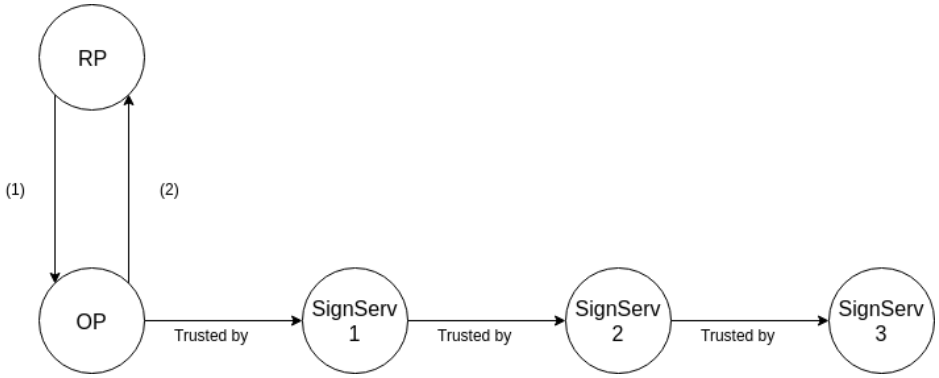


Figure 9 *RP requests Entity Statement from the specified OP*

Now the RP will try to follow trust chains that have been suggested by the OP, trying to find any Trust Anchor. In its search, the RP can encounter Signing Services that at some point only refer to themselves, this is considered the end of a chain, it can encounter a Signing Service that is unwilling to sign an Entity Statement, or it can find one that has authority hints that go up the chain. In both the first and the last case, the RP will have to determine if they trust the anchor, and if not, how to continue the search for a trusted party. This means the RP will attempt something like the following, excluding the requests made to any Signing Services that are part of an untrusted chain.

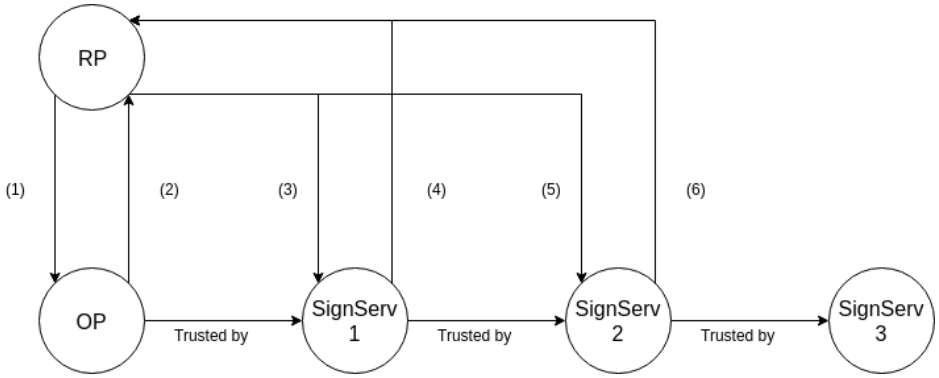


Figure 10 *RP keeps requesting Signing Services' Entity Statements, until a trusted party is found*

After encountering a trusted Signing Service at step 6, the RP will conclude that if the OP is under the umbrella of a trusted Signing Service, it can trust the OP as well - meaning the flow as noted above will begin.

With the GÉANT project[6] wrapping up, the specification going into public review[7], and the call for testing, OpenID Federations is giving off signals that it is ready to be finalized.

[6]https://wiki.geant.org/display/gn42jra3/T3.1A+OpenID+Connect+Federation
[7]https://openid.net/2018/06/08/public-review-period-for-openid-connect-federation-specification-started/

Chapter 3

Decomposition of work

In this chapter, we will discuss how the System Under Test was set up, what the specification entailed and how the wording of the specification relates to the model that was written to represent it. We also show how the model touches upon the in and output of the System Under Test, and how the design decisions of the model were made, both of the Signing Service and the Relying Party, which need a slightly different design in their communicating nature. Finally, there is a description of work that has to be done to be able to execute tests based on the model in the Axini Modeling Suite.

3.1 Setting up a testing environment

To do anything related to Model Based Testing, there has to be something to test: a System Under Test (SUT). As there are three components to the implementation, a choice had to be made. Eventually the Signing Service, which is an implementation of the Trust Anchor, was chosen as the first component to be verified. This component is very verbose in nature: it will sign whatever it is familiar with, while gracefully refusing anything else. This fits very well into the Input-Output compliance model that the Axini testing suite is based upon.

While setting up the example environment, it becomes quite clear that the software is not immediately ready to use. The root of this problem lies in part in the versioning of libraries used, which are written by the same developer. For every library, it was required to find the right version in git, and then install those manually. This took a few weeks due to the number of libraries and the number of versions to try. While the versions of the required libraries had a specified minimum version, these were not properly defined in the libraries themselves, and components were moved between libraries, invalidating their references within the code that uses them. Most of these libraries were stored in git, and so using an approximation with the date as a leading factor combined with simple trial and error, the (presumed to be) right commit numbers could be determined.

While there are some scripts that should set up (part of) an example environment, there is very little written down on how to get the implementation to work in a normal setting. The biggest steps in understanding came during an attempt to recreate the software. Eventually, the signing service was set up, and while the OP and RP made progress towards a working environment, these did eventually not reach that point.

3.2 Verifying the Signing Service

In order to verify if the OIDCfed reference implementation implements the specification correctly, there has to be a software implementation of the process visualized in Figure 2. This software has to be able to generate test cases from the model, execute the tests, and verify them against the expected outcomes. It has to be able to ingest changing models, while being able to communicate with all components of the System Under Test. The Axini Modeling Suite (AMS), based on the principles of ioco and other academic work by Jan Tretmans, does these things, generating tests based on definable strategies and providing a way of communicating with multiple entities using a technology agnostic framework.

3.2.1 Functional Behaviour

With combination of the adapter and the model being the vehicle for testing the implementation, there is the question of what parts of logic go where. Axini voiced their preference for adding the biggest amount of logic and decisions to the model. On the downside, the model has a limited amount of functionality to its disposal, while many of the checks are in a specific format (e.g. cryptography or encoding related).

In the written specification, the following is written on the topic of https endpoints that have to be made available[6]:

```
Given the entity identifier, the endpoint of the federation API
   endpoint of the entity is easily derived. All entities that are
   expected to expose entity statements about themselves or other
   entities, MUST implement a Federation API.

The federation API endpoint of an entity is resolved from the entity
   identifier. The Federation API endpoint is found using the Well
   known URIs specification, with the suffix openid-federation. The
   scheme, host and port is taken directly from the entity identifier
   combined with the following path: /.well-known/openid-federation.

The Federation API is an HTTP API that may support multiple operations.
    Fetching entities is one of the operations, and the only one that
   all entities are REQUIRED to support. All the other operations is
   OPTIONAL. The op (operation) parameter may be used to define new
   operations in a future version of the specification or a local
   deployment which have agreed upon additional functionality.

While all operations in the specification make use of a GET request,
   other operations may choose to use other HTTP methods. If the op
   parameter is left out, it is considered to be a fetch entity
   statements request. Unless otherwise mentioned or agreed upon,
   requests to the federation API does not need to be authenticated.
```

Figure 11 *Formal specification of OIDCfed's web endpoint*

From this we can infer that there exist a number of operations that the signing service can implement, with only one being mandatory: 'fetch'. When inspecting the implementation that is under test, it becomes clear that not all the non-required operations are implemented, and thus the model below only consists of the 'fetch' operation. This operation should return a signed statement about the requested subject, if the signing service is willing to do so. This fetch operation has two parameters with regards to who signs what: a mandatory issuer and an optional subject. If the subject is left out, it will be equal to the issuer. If the subject is included, it can be anything, including the issuer.

This is worded as follows[6]:

```
The request MUST be an HTTP request using the GET method to a resolved
   federation API endpoint with the following query string parameters:

op
OPTIONAL. If not present MUST be treated as fetch.
iss
REQUIRED. The entity identifier of the issuer from which you want an
   entity statement issued. Because of the normalization of the URL,
   multiple issuers may resolve to a shared federation API. This
   parameter makes it explicit exactly which issuer we want entity
   statements from.
sub
OPTIONAL. The entity identifier of the subject for which you would like
    an entity statement issued. If this parameter is left out, it is
   considered to be the same as the issuer, and would indicate a
   request for a self issued statement.
```

```
aud
OPTIONAL. The entity identifier of the requester. The issuing entity
   may choose to include this parameter to form the entity statement
   specifically for this target, in which the aud claim also should be
   present in the entity statement it self.
prefetch
OPTIONAL. If left out, it is assumed to be false. If set to true, it
   indicates that the requester would like the API to prefetch entity
   statements that may be relevant for consumer to establish a trust
   chain, and may save the consumer from performing additional API
   requests.
```

Figure 12 *Formal specification of the Signing Service's 'fetch' operation*

From these parameters, we can incur a few scenarios: a request can have either one or no subject, a request can be valid or not, and a signing service can be willing to sign or not.

'aud' is a parameter which is both optional in the request and the implementation, only being necessary when the statement that is issued is actually tailor-made for the audience at hand. As this is not supported by the reference implementation, and there is no way to systematically check if the missing output is unjustifiably missing - after all, both being present and being absent can be either good or bad based on what the SUT has decided - it is not considered in the model below.

'prefetch' indicates a preference of the requesting party that might or might not be taken into account. As another unsupported operation from the point of view of the reference implementation, it is not considered in the model below either.

The considerations as mentioned result in the following model:

```
external "Anchor"

process ("fetch") {
  channel ("Anchor") {
    stimulus 'get_valid_cert', '_withsub' => :boolean
    stimulus 'get_invalid_cert'
    response 'valid', '_iss' => :string, '_sub' => :string
    response 'invalid'
  }

  timeout 10.0

  choice {
    o {
      receive 'get_valid_cert', constraint: '_withsub == true'
      send 'valid', constraint: '_sub != _iss'
    }
    o {
      receive 'get_valid_cert', constraint: '_withsub == false'
      send 'valid', constraint: '_sub == _iss'
    }
    o {
      receive 'get_invalid_cert'
      send 'invalid'
    }
  }
}
```

Figure 13 *Model corresponding to OIDCfed's 'fetch' operation*

This model is equal to the following visualization.

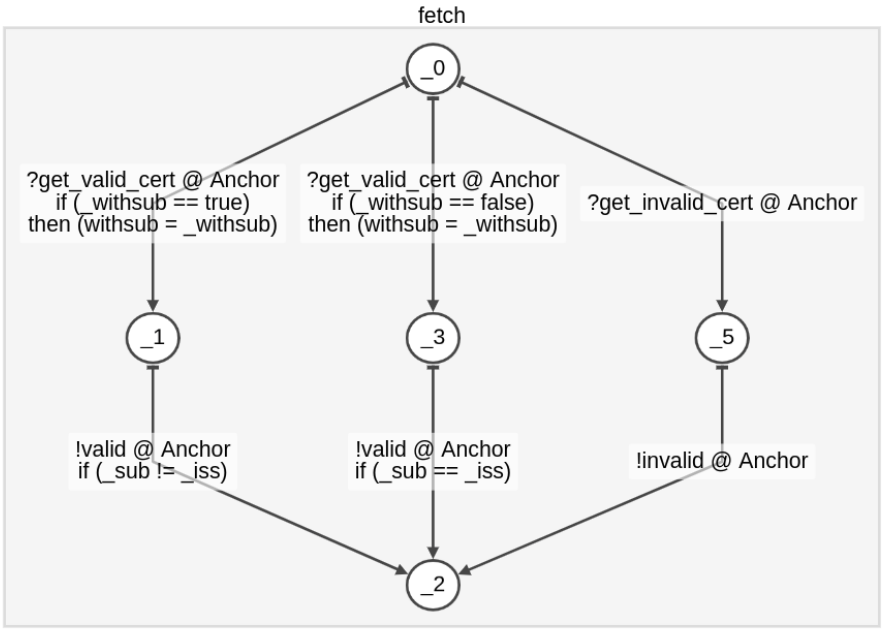


Figure 14 *Visualization of the model of the 'fetch' operation*

The valid and invalid paths are clearly visible, while the use or lack of 'sub' is determined by the use of parameters. In the case of '_withsub' being true, the adapter sets it to a different value than the issuer to mimic usual behaviour. After all, the self-signing capability is already tested with the sub not being present. This model is quite small, as the adapter decodes the response of the web endpoint, looking like this:

```
"eyJhbGciOiJSUzI1NiIsImtpZCI6ImFFUllRUzFNVUZsdFJWVndUMXAwZFVkeGRFdHhUb
EpyT1U5cGIxQnhNVGM0V2tsT1VtSktkVkZpTkEifQ.eyJpc3MiOiAiaHR0cHM6M0E6MkYl
MkZwdXQubmlraGVmLm5sJTNBODQ0MyUyRm5pay1hbmNob3IiLCAiaWF0IjogMTU2NDE1Nj
I1OSwgImV4cCI6IDE1NjQyNDI2NTksICJtZXRhZGF0YSI6IHsibWV0YWRhdGEiOiB7Im9w
ZW5pZF9jbGllbnQiOiB7fX19LCAic3ViIjogImh0dHBzJTNBJTJGJTJGcHV0Lm5pa2hlZi
5ubCUzQTg0NDM1MkZuaWstYW5jaG9yIiwgImp3a3MiOiB7ImtleXMiOiBbeyJrdHkiOiAi
UlNBIiwgInVzZSI6ICJzaWciLCAia2lkIjogImFFUllRUzFNVUZsdFJWVndUMXAwZFVkeG
RFdHhUbEpyT1U5cGIxQnhNVGM0V2tsT1VtSktkVkZpTkEiLCAibiI6ICJyTzd0WGdCYUtx
a0lkLWgwMlVjMnlGeWJXRGVXajlQclFzeGhsalhtYlBWa0pNOHJERElSRU9faDkzM1VUcn
pfTWZ0aDFCR1BQOEYtYXBYUVFxZE1iV193MEhEMzAwWGJiamJPbmJVM21pRmF4cGdObjBo
dDBDS1MxQ0FkUVU5VUhXT1RmWEdpRzJnTjhkNzJrSXhydGY1M2lGdEtCOE5aUi1RWnkzWU
FjNG4yNG9yRUFBR29COXR5R1V2WFYtaWNBLXBNSk1CVUVDdkkteUJSZXBHU3JQTThXUnF1
eG1Ta1NhTkF6OU94R05fc0ZqMHRSMkJvOUVkVF82dVhaZnNHRDdUai12ZmtEdDJWNXZ5NV
8wdHprTFh1T2VTUjVsVk41OGx0R1BpTzZfZEdURXJZUU16UFBQLUR5YzAyWnlEMkV0eHFo
Q2RPZGFwaWx2Tk5KNWVZeXciLCAiZSI6ICJBUUFCIn0sIHsia3R5IjogIkVDIiwgInVzZS
I6ICJzaWciLCAia2lkIjogIlNFaHNkMVpFVUVSdlJub3dTa3hSYVVodU1tcG9jME5rVlRo
MU5WSnRNRFkyWm1ad2JFWndXbTR5U1EiLCAiY3J2IjogIlAtMjU2IiwgIngiOiAiVjBVZG
dVd19ESnVEOTZ1dWp4ODFYSlJQcXFLZ2VjRVJBRUFPcHUtVVRKOCIsICJ5IjogIkp3RDM1
SnhGVWdrSjRmQ2NkME5HSzR5VVE4UFQ1OHJJUW9JMkhCZFpkbWcifV19LCAia2lkIjogIm
FFUllRUzFNVUZsdFJWVndUMXAwZFVkeGRFdHhUbEpyT1U5cGIxQnhNVGM0V2tsT1VtSktk
VkZpTkEifQ.RFLuDorovAA0_27qVLUzQfmiIiTN8tjlxMtbbVRkjHPYancezC7dr4vQtVn
353DzthIWsHtysfNP_Tubtt446fvzlXFqrUoOGF0ClCTbLeBo1Yv0paQRFo8DudP6Q5Dce
TBcwFu7RcPtoJGOYYSTGFIV90zQZvYAHHlx-dACUqPafUsXjYta2tBXBVg5BsKEHTZ9FOs
CqUP8p20GINEizAuwihhYikHLakhhnxQGbrrx7etS4ETWf3jeRZGoxLTTJqBdC_8f9geHW
hGfEEmQVE_3RfrOyMYvaD2pEz6rpwWbUpyots-CLhLn5-uC0EU-XcodUN_F2ZUVQlKqKGt
mxQ"
```

Figure 15 *Example output of the 'fetch' operation*

This is a Java Web Token (JWT), containing three parts: header, payload and signature. The header and payload can be decoded as follows.

```
{
  "alg": "RS256",
  "kid": "aERYQS1MUFltRVVwT1p0dUdxdEtxTlJrOU9pb1BxMTc4WklOUmJKdVFiNA"
}
.
{
  "iss": "https%3A%2F%2Fput.nikhef.nl%3A8443%2Fnik-anchor",
  "iat": 1564156259,
  "exp": 1564242659,
  "metadata": {
    "metadata": {
      "openid_client": {}
    }
  },
  "sub": "https%3A%2F%2Fput.nikhef.nl%3A8443%2Fnik-anchor",
  "jwks": {
    "keys": [
      {
        "kty": "RSA",
        "use": "sig",
        "kid": "
           aERYQS1MUFltRVVwT1p0dUdxdEtxTlJrOU9pb1BxMTc4WklOUmJKdVFiNA",
        "n": "rO7tXgBaKqkId-
           h02Uc2yFybWDeWj9PrQsxhljXmbPVkJM8rDDIREO_h933
        UTrz_Mfth1BGPP8F-
           apXQQqdMbW_w0HD300XbbjbOnbU3miFaxpgNn0ht0CKS1CAd
        QU9UHWOTfXGiG2gN8d72kIxrtf53iFtKB8NZR-
           QZy3YAc4n24orEAAGoB9tyGUvXV
        -icA-pMJMBUECvI-
           yBRepGSrPM8WRquxmSkSaNAz9OxGN_sFj0tR2Bo9EdT_6uXZf
        sGD7Tj-vfkDt2V5vy5_0tzkLXuOeSR5lVN58ltGPiO6_dGTErYQMzPPP-
           Dyc02ZyD
        2EtxqhCdOdapilvNNJ5eYyw",
        "e": "AQAB"
      },
      {
        "kty": "EC",
        "use": "sig",
        "kid": "
           SEhsd1ZEUERvRnowSkxRaUhuMmpoc0NkVTh1NVJtMDY2ZmZwbEZwWm4ySQ",
        "crv": "P-256",
        "x": "V0UdgUw_DJuD96eujx81XJRPqqKgecERAEAOpu-UTJ8",
        "y": "JwD35JxFUgkJ4fCcd0NGK4yUQ8PT58rIQoI2HBdZdmg"
      }
    ]
  },
  "kid": "aERYQS1MUFltRVVwT1p0dUdxdEtxTlJrOU9pb1BxMTc4WklOUmJKdVFiNA"
}
```


Figure 16 *Decoded JWT token, showing a self signed Entity Statement*

Clearly, there are more properties than the issuer and subject. This does not, however, mean that they are necessarily worth verifying, as they might be optional, or their values might not be exactly specified by the specification. 'iat' (issued at), for instance, is simply an indication of when the message what created, and 'exp' (expires at) means the issued statement is no longer to be trusted after that time. The specification only says they should be there, and that the consuming services should verify the expiry time has not yet passed. With no defined time frame limitations, however, any value in exp is hardly

verifiable. The same goes for metadata, which is a configurable and optional field, meaning that if it is not there, it is no problem, and if it is there, it can contain a great number of different values. The one thing that would still be relevant to check is if the signing key from the JWT header is indeed in the JWKS of the entity statement. This can only be checked for self-signed statements, as other entities will not have the same key under normal circumstances. A straightforward way of verifying the signing key is adding another constraint to the model, concluding all required and verifiable parts of the signing service.

```
external "Anchor"

function('contains', [[:string], :string] => :boolean) {|list, entity|
    list.include? entity}

process ("fetch") {
  channel ("Anchor") {
    stimulus 'get_valid_cert', '_withsub' => :boolean, '_signing_kid'
        => :string, '_jwt_kids' => [:string]
    stimulus 'get_invalid_cert'
    response 'valid', '_iss' => :string, '_sub' => :string
    response 'invalid'
  }

  timeout 10.0

  var 'withsub', :boolean

  choice {
    o {
      receive 'get_valid_cert', constraint: '_withsub == true &&
          _signing_kid in _jwt_kids', update: 'withsub = _withsub'
      send 'valid', constraint: '_sub != _iss'
    }
    o {
      receive 'get_valid_cert', constraint: '_withsub == false && !(
          _signing_kid in _jwt_kids)', update: 'withsub = _withsub'
      send 'valid', constraint: '_sub == _iss'
    }
    o {
      receive 'get_invalid_cert'
      send 'invalid'
    }
  }
}
```

Figure 17 *Model with enhanced post-conditions, showing key id validation*

The _jwt_kids and _signing_kid will be extracted by the adapter.

3.2.2 Error Handling

In case of any error that occurs during a request, there has been a format specified to represent these errors. Consider the following excerpt from the specification[6].

```
If the request was malformed, or some error occurred during processing
    of the request, the following standardized error format should be
    used regardless of the operation specified.

The HTTP response code MUST be something else than 200, giving an
    indication of the type of error. The response body MUST be a JSON
```

```
   object containing the claims below and the content type MUST be set
   to "application/json".:

op REQUIRED.  Which operation was the request processed as.
error  REQUIRED.  The error code.
error_description  REQUIRED.  A human readable short text describing
   the error.
```

Figure 18 *Formal specification of OIDCfed's error responses*

These conditions too, can be checked in the adapter, based on which a response is sent to the model - in which case the model will look like the ones above. A more reasonable approach would be to do more verification in the model. In any case, an error has to be forced. The adapter can decide on how to do so - and potentially decide on certain implementation specific conditions. The response code could for instance be checked based on what error is forced to occur. In any case, it is possible to check the header and the body of the error message.

```
external "Anchor"

process ("fetch") {
  channel ("Anchor") {
    stimulus 'get_valid_cert', '_withsub' => :boolean, '_signing_kid'
       => :string, '_jwt_kids' => [:string]
    stimulus 'force_error'
    response 'valid', '_iss' => :string, '_sub' => :string
    response 'invalid', '_headers' => [:string], '_fields_present' =>
       [:string]
  }

  timeout 10.0

  var 'withsub', :boolean

  choice {
    o {
      receive 'get_valid_cert', constraint: '_withsub == true &&
         _signing_kid in _jwt_kids', update: 'withsub = _withsub'
      send 'valid', constraint: '_sub != _iss'
    }
    o {
      receive 'get_valid_cert', constraint: '_withsub == false && !(
         _signing_kid in _jwt_kids)', update: 'withsub = _withsub'
      send 'valid', constraint: '_sub == _iss'
    }
    o {
      receive 'force_error'
      send 'invalid', constraint: '"Content-Type: application/json" in
         _headers && "op" in _fields_present && "error" in
         _fields_present && "error_description" in _fields_present'
    }
  }
}
```

Figure 19 *Model containing the requirements related to error responses*

3.3 Verifying non/lesser verbose parts of OIDCfed

While the Signing Service that in this work is a clear communicating system, the OP and RP of the OIDCfed environment hide their behaviour, only showing the final decision (does the RP, or does it not

trust the OP). One way to test this is to create situations where certain aspects of the decision making process are altered and highlighted, hoping the right choices are made for the right reasons. Another, more invasive, approach would be to add points of control and observation (PCO) to the system under test.

The easiest way to go about these PCOs, is to emulate the behaviour the components usually show. This means rewriting parts of the implementation, or weaving logic into the existing components, adding channels to the model. While this did not fit into the time line of the research, an example model that should use these principles has been developed to aid further research.

```
external "Anchor"
external "OP"
external "RP"

process ("Login") {
  channel ("Anchor") {
    stimulus 'add_hint'

    stimulus 'prepare_refuse_signing_trusted'
    stimulus 'prepare_accept_signing_trusted'
    stimulus 'prepare_accept_signing_untrusted'

    response 'received_fetch'
  }

  channel ("OP") {
    stimulus 'show_auth_hint'
    stimulus 'show_no_auth_hint'

    response 'received_fetch'
  }

  channel ("RP") {
    stimulus 'initizalize_login'
    response 'retrieved_metadata'
    response 'verified_signature'
    response 'op_decision', '_trusted' => :boolean
  }

  timeout 5

  var 'should_trust', :boolean

  receive 'initizalize_login', on: 'RP'

  choice {
    o {
      receive 'show_auth_hint', on: 'OP'
      send 'received_fetch', on: 'OP'
      send 'retrieved_metadata', on: 'RP'
      send 'verified_signature', on: 'RP'

      repeat {
        o {
          receive 'add_hint', on: 'Anchor'
          send 'received_fetch', on: 'Anchor'
          send 'retrieved_metadata', on: 'RP'
          send 'verified_signature', on: 'RP'
        }
```

```
        o {
          choice {
            o {
              receive 'prepare_refuse_signing_trusted', on: 'Anchor',
                  update: 'should_trust = false'
            }

            o {
              receive 'prepare_accept_signing_trusted', on: 'Anchor',
                  update: 'should_trust = true'
            }

            o {
              receive 'prepare_accept_signing_untrusted', on: 'Anchor',
                   update: 'should_trust = false'
            }
          }

          send 'received_fetch', on: 'Anchor'
          send 'retrieved_metadata', on: 'RP'
          send 'verified_signature', on: 'RP'

          send 'op_decision', on: 'RP', constraint: '_trusted ==
              should_trust'

          stop_repetition
        }
      }
    }

    o {
      receive 'show_no_auth_hint', on: 'OP'
      send 'received_fetch', on: 'OP'
      send 'retrieved_metadata', on: 'RP'
      send 'verified_signature', on: 'RP'

#       Ask trusted anchors?
#       From spec:
#       The information found in the authority_hints is critical in
    order to dynamically discover the trust chain.
#       If such hints are not present, the RP may fall back to fixed
    configured trust roots to ask for entity statements.
      send 'op_decision', on: 'RP', constraint: '_trusted == false'
    }
  }
}
```

Figure 20 *Model of the communication and behaviour of the RP, including 'controlling' signals*

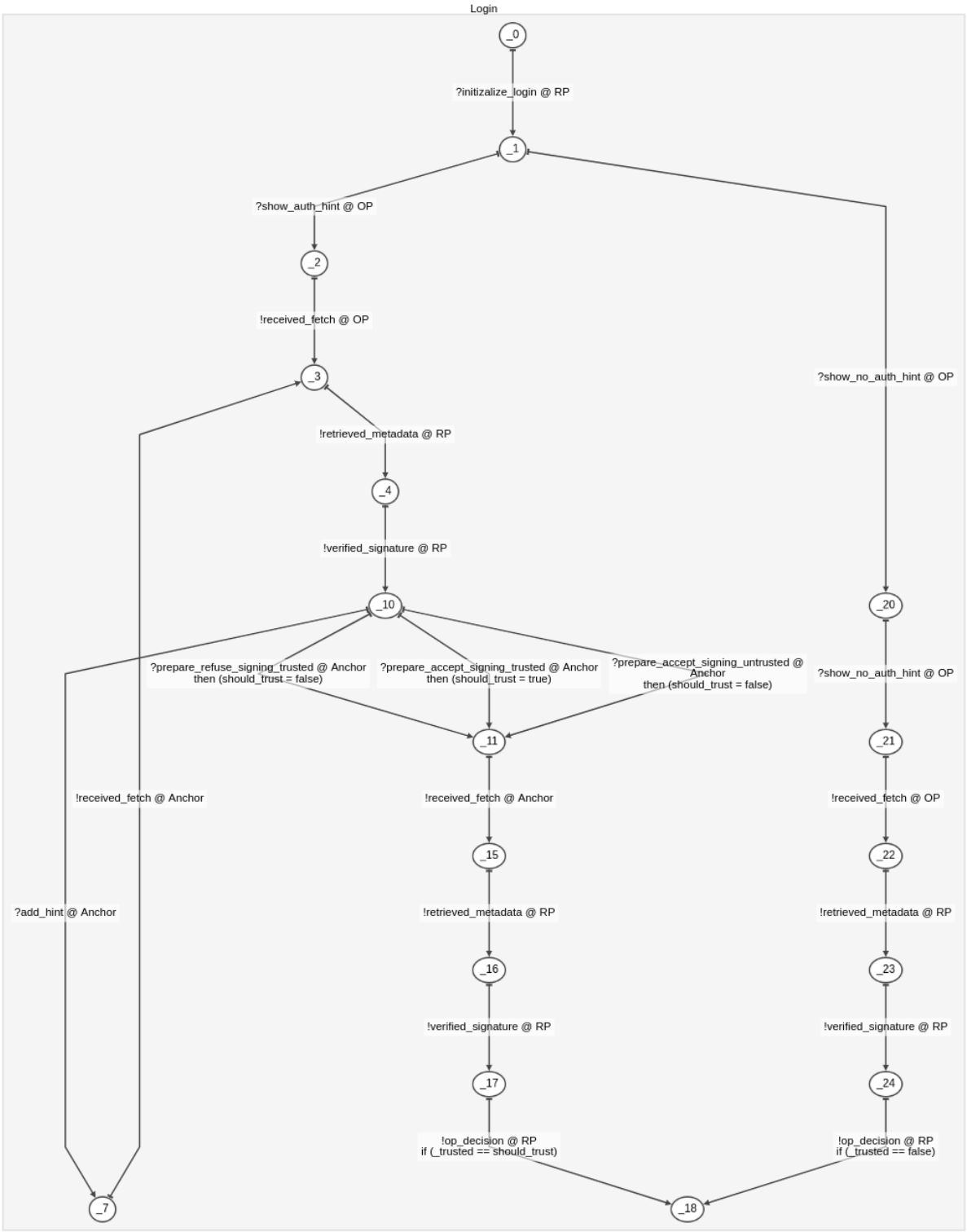


Figure 21 *Visualization of the RP model*

The OpenID Provider (OP), as seen in the above model, does not have any specific federation behaviour as such. Rather, it publishes information that is used to make decisions about the OP by the Relying Party (RP). Also, when looking at the written model, there is a piece of behaviour that is difficult to verify, and might work best as an option of sorts: what to do when there are no authority_hints. According to the specification, the RP *may* show certain behaviour, seeing if by some coincidence, signing services it knows about, trust the party it wants to verify. If it does not, this is also fine.

With the reference implementation having various issues regarding its setup, it was not possible to test the model that has been developed for these components. With very limited documentation,

attempts to set up the last components resulted in errors such as *"String [...] is not an integer"*, which in this case meant a wrong configuration setting. Eventually, not being able to solve dependencies made the testing of the OP and the RP an unreachable target for this research. Even if the setup had been successful, the integration of PCOs would have been a risk to the viability of this part of the research, especially with regards to the time that has been reserved for the full project.

3.4 Adapter Design

As a modular part of the Axini Modeling Suite, adapters can be set up in close to any way the developer desires, as long as it communicates in the expected way - using websockets. With the Signing Service being a web service, this research enjoyed the luxury of being able to use the adapter library that has been developed within Axini, meaning getting most of the communication 'for free'. As mentioned, an example adapter using this library can be seen in appendix 1. The design of the adapter is inherent to the environment that is used to implement it. It is possible to do dynamic function calls, depending on the programming language, or to approach this with a single point of entry. As the adapter was very limited in functionality, and dynamic function calls can be a risk with regards to software integrity as some of behaviour can become unchecked, the adapter that can be seen in the appendix uses a simple switch-like structure to execute the wanted behaviour.

As the model for the RP and OP are more invasive, it might not be possible to use the Ruby libraries that Axini has built. In stead, there might be a need to go into the code of the SUT - which is Python in this case - to make it communicate, or to develop a proxy that more closely observes the behaviour that the SUT is showing. Regardless, according to Axini, any adapter should be, to quote a famous expression, as simple as possible, but not simpler.

3.5 Executing Model Based Tests

In order to handle stimuli and responses, which do not necessarily entail an action that is understood by the SUT, and to handle setting the right environment for the SUT, the Axini platform works with so-called adapters. In the case of the Axini platform, they are stand-alone entities that talk to the platform. The adapter then 'converts' these messages into actions, and vice versa. In order to get an idea of the adapter, see appendix 1.

When invoked, the test manager will find all paths that have to be tested and execute stimuli based on that, verifying the results. The first runs contained a function that would automatically try to infer if it had already tested the SUT with or without the subject, but this did not work out as expected, and thus the with and without subject became two separate paths.

Chapter 4

Results

4.1 Consistency in metadata storage

Quite quickly into the research, I ran into a problem with the signing service. There is a distinction between statements that are being made about itself, and statements that are made as another issuer, in the sense that they are stored differently. This distinction made signing fail in some cases. As this problem blocked my from testing the software any further, I decided to fix it and place it in the official repository[1].

4.2 Release of self-signed entity statements

After modeling the Signing Service (Trust Anchor), the first inconsistency with the specification arose. In the setting up of this component, a derivative of the supplied configuration had been installed. Using model based testing, the lack of self-signed entity statements came up. The specification says the following on the matter: *"A trust anchor must publish a self-signed entity statement about itself"*. Our model reflects this requirement by testing if the Signing Service supports requests with only an issuer, which is supposed to return a statement where the subject is equal to the issuer. The editor has been notified of the problem, but there have not been any public fixes for this so far.

4.3 Functioning of the RP and OP

While the model for the interaction between the RP and the OP/Signing Service have been developed, the software has not reached a testable state. This means that while it is hard to verify its correctness, it is possible to say that the software that surrounds the RP and OP implementation is not consistent enough to get a stable environment. There is a model that represents what is in the specification, but has not been used and therefore might lack improvements that come up in practice.

Complementing the model is the knowledge that has been obtained on how this system can be tested, in particular the design where PCOs should be used to observe and influence fine grained behaviour with the aim to highlight certain behaviour. This means that there is even less manual work involved in testing these systems: if there are no PCOs, that would mean having to set up all sorts of different environments by hand (or at least write scripts to do so), which is an exponential amount of work for each small change that can occur somewhere in the chain.

4.4 Bad error handling

The specification contains a section on how errors should be returned. Besides the written specification, there is the following example:

[1] https://github.com/rohe/fedservice/commit/e993c36ac4329059682ad66a9befbe61dcf8b60c

```
400 Bad request
Last-Modified: Wed, 22 Jul 2018 19:15:56 GMT
Content-Type: application/json

{
  "operation": "fetch",
  "error": "invalid_request",
  "error_description":
    "Required request parameter [iss] was missing."
}
```

Figure 22 *Specified example of OIDCfed's error representation*

This output shows what part of the specification has been violated, therefore it should be possible to recreate the scenario. To refer back to *ioco*, given input 'missing iss parameter on operation fetch', the Signing Service should give the above output. Given the mentioned input, the current reference implementation will result in a non-conforming output.

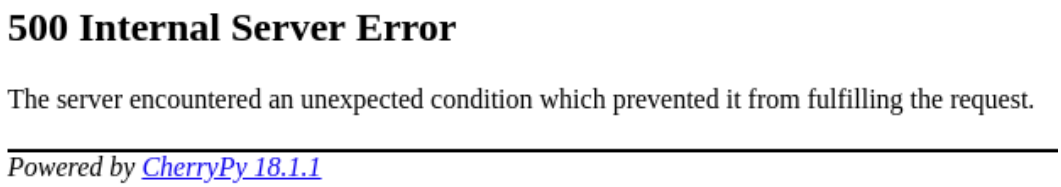


Figure 23 *Error response presented by reference implementation*

Clearly, this is HTML-formatted, and not a JSON object with a corresponding header as required by the specification, nor does it contain the three fields containing related messages. Besides its content, the usage of the 500 Internal Server Error status code is incorrect, especially with the example showing the better fitting 400 Bad Request.

4.5 No support for non-required operations

During the modeling of the signing service, the source code of the reference implementation showed that it only contained required operations. This is not an error in the software itself, but makes the development of a full model harder as well, besides decreasing completeness as a proper implementation example.

Chapter 5

Discussion

5.1 Research Questions

Given these findings, we can answer the questions asked before:

RQ1. *Is it possible to create a model from the written specification that can be used to check the reference implementation?*
We have shown that it is possible to map OIDCfed's Signing Service behavior on an input-output level.

RQ2. *Does the usage of* MBT *result in the finding of errors in the reference implementation?*
We can verify that under specified conditions, expected behavior takes place. We also found, though, that in order to verify internal behavior, we have to create environments where these aspects pop out, and assume that the right behavior is not coincidental. This can be either done manually by setting up environments, or by using PCOs.

RQ3. *To what extent are the errors that are found by* MBT *violations of the specification?*
Given the results shown above, we can say that the reference implementation, combined with the default configuration, does not always work as the specification prescribes. Especially the incorrect error representation is a clear violation of the specification, while the others are more indirect, creating a violation of the specification because of incorrect configuration.

RQ4. *How does the usage of* MBT *compare to manual testing in terms of incorrect behavior detection?*
Given the fact that MBT found issues, even though the testing that had happened before was done by experienced people, it is safe to say that Model Based Testing results in more findings than manual testing. Please do note the discussion below.

5.2 Validity of the result of RQ4

Whether or not using MBT results in more findings of incorrect behavior than manual testing, is of course depending on a number of factors. Not the least is that every finding - as long as it's reproducible - is also discoverable when testing manually. This is where a fundamental problem of manual testing arises: it is strongly dependant on the creativity and physical limits (such as time) of the tester. There might be structural methods to minimize the amount of creativity needed, but eventually it all comes down to the person thinking about what is needed to uncover unwanted behavior.

This means that somehow, the comparison between the manual testing and the Model Based Testing has to have substance. In an attempt to do so, the following can be said about the manual testing that has been done.

- With the reference implementation, unit-tests have been developed
- The developer of these tests is the same person that wrote the implementation and the specification - suggesting knowledge of the material
- The developer of these tests has built the test suite that later became part of the official OIDC certification process - suggesting knowledge of unit testing, and in particular systems like the one under test
- At least one engineer with experience in federated trust systems has manually set up the reference implementation, and has not found the issues

- When I started this research, I had close to no knowledge of the System Under Test, nor had I used it before

Given these points, we can infer that someone that has no experience with the matter at hand can find issues that experienced engineers with knowledge of the systems had not yet discovered, and therefore, we can say that using MBT results in more, or at least different findings. Potential explanations for this could be systematic interpretation of what a system is supposed to do. This works for OIDCfed as it has such a specification. Another explanation might be that someone that creates the model only has to think about what the system has to do, and not what the system could do that it's not supposed to do. After all, it is the test suite that should uncover bad behavior.

5.3 Is the model a valid representation of the specification

In order for Model Based Testing to add value to the verification process, the model has to be a trusted representation of the written specification. This trust comes from two sources: on the one hand, in the case of OIDCfed, there have been specifically defined inputs and outputs. It is this in and output that is verified, and thus in the very least, the example that is given in the specification, should with the correct setup, yield the response that is formulated in the specification. This is because the Axini platform is testing the before mentioned ioco relation. On the other hand, any error that is found with the defined model creates an evaluation point for the model itself. After all, a test failing under circumstances that should have resulted in success will mean an evaluation of the whole environment, including the model. Should the model not have been correct, then this will be fixed after such an incident.

The issues with the implementation, can and have been manually verified against the specification, and/or with the help of the specifications editor. This means that the model that has been developed, and the tests that have been generated are *sound*. *Exhaustiveness* of the tests that are inferred from the model is harder to determine. By their nature, cryptographic protocols are not unbreakable, the process of breaking them should 'simply' take so long, that breaking them is not viable. This means signatures can be correct for the wrong reasons, and the decisions made based on them can be right for the wrong reasons. These factors have not been taken into account in the modeling. Some parts of the specification, as they were not implemented, and no behaviour was expected per se. This means that the model and its tests are, while they are *sound*, not *exhaustive*.

5.4 MBT versus Unit-Testing

As with most engineering approaches, MBT and Unit-Testing both have their pros and cons, implying the ability to comparable the two. The question is, however, if this is a fair comparison. Certainly, the usual approach to testing with MBT will most likely be different than testing with Unit-Tests, but that does not mean one set of technology is necessarily better than the other. The underlying technology stack being able to do more-or-less the same, does not also mean that the different approaches result in the same outcome. As shown before, the systematic approach of MBT, using the written specification in a structured manner has lead to results that had gone unnoticed before, even with Unit-Tests that were developed by an experienced test writer. Even so, different adapters might result in different findings. The adapter that has been developed for this research stayed on the surface, but this approach might be quite a bit harder for the not-so-verbose parts of OIDCfed - which could probably use a more encapsulating approach to verify if it is showing the right behaviour. The best way to describe the added value of MBT is by highlighting the versatility that the model adds. As long as it is a source of truth about the software's behaviour, it can be used to verify with different approaches, depending on the eventual executor.

Chapter 6

Related work

Approaches towards the verification of distributed (security) systems often relies on a form of model extraction, combined with model checking. An example of related software being verified this way is the checking of various SSH implementations[7]. This approach has more of a Model Checking basis than a pure Model Based Testing one, where a model was extracted based on the behaviour that the system shows. In order to combine this Model Learning to Model Based Testing, there has been effort related to using Model Learning as input for Model Based Testing - meaning Model Learning can be used to verify the software using one of two methods.[8] This way of verification requires specific implementations to support model learning. In production security software, these additions can be unwanted due to the unnecessary risk that they pose, exposing behaviour of a closed system.

Another approach to verification is extracting models using static analysis. Ironically, software does not necessarily have to work in order to analyze it, should the analyzer accept the fact that it does not know everything about the software, such as external references, working even with distributed environments.[9] This too, is based on Model Checking. With the basis of this specification is not directly dealing with the input, but rather combining multiple other specifications with regards to encoding and cryptography, specifying communications between components, this kind of extraction has the risk of becoming too complicated for what we are trying to achieve, while becoming quite hard to verify against a written specification. By using MBT, we can determine the relation of the software to the specification. An example of MBT being applied to communicating software where trust is important is the verification of passport protocols.[10]

OpenID Connect Federations can be seen as a system of systems, which come, due to their distributed nature, comes with challenges and certain situational awareness. Related work in this field mainly comes from maritime security in the north sea.[11] Model Based Testing distributed trust systems is not very common yet, mostly relying on inferred-model checking in stead of testing.

With regards to Model Based Testing, there is a lot of work happening around Jan Tretmans at the University of Nijmegen, some of which has already been referenced in the forms of ioco[4] and passport verification.[10] Initiatives such as SUMBAT explore the application of MBT in embedded systems. Software to do Model Based Testing focusses around projects like TorX derivatives and the Axini Modeling Suite.

Related to the validity of the developed model are soundness, completeness and exhaustiveness.[4] These properties are significant to the model, and therefore relevant to the results of the tests that are being generated from the model. With ioco being one way of signifying compliance, there are also other, more strict ways of signifying the correctness of software such as trace equivalence (which includes invisible transitions), or bisimilarity.

Chapter 7

Conclusion

In this research, it has become clear that OIDCfed is a very suitable candidate for modeling. The specification contains clear instructions on input and output, where applicable. Some features in the specification are optional and not present in the reference implementation. In these cases, a feature flag can be a solution for specific implementations.

The notion that the OIDCfed software was reaching its final state has probably come too early. As mentioned, some optional behaviour was not always implemented, while a reference implementation might be a safeguard for ambiguous descriptions, and the fact that its environment is more volatile than what would be expected of such reference software, making the implementation harder to use and therefore harder to test.

With the models that have been written and executed, issues with regards to error representation and configuration came to light. These have been passed on, and will be fixed. With the coming of new implementations in different programming languages, there is assurance that the findings of this research will live on in the new implementations.

After using Model Based Testing to verify OIDCfed, it is possible to infer that MBT does bring advantages over normal testing, and especially unit testing. MBT adds structure that is strictly bound to the specification, making the tester (or the model writer) verify the System Under Test in such a way that it will also find any problems that might have been forgotten by a unit test writer, regardless of the experience of the tester. On the other hand, verifying internal behaviour that can be a very important part of software such as OIDCfed, needs additional work over a relatively simple black box testing approach, shining light on some of its internal behaviour by adding Points of Control and Observation.

Besides the testing itself, modeling can help with finding ambiguity in the specification. See for instance the prefetch option in the Signing Service. This systematic way of modeling sheds a new light on specific pieces of text, not unlike 'Rubber Ducking'[1].

In short, MBT is a very valuable tool when it comes to the verification of specified systems, as long as it is able to see 'inside' parts of the system when needed. In the case of OIDCfed, it has not been possible to verify the system as a whole, due to a lack of working software, while everything is in place to test it.

7.1 Future work

7.1.1 On OIDCfed Development

In order to do meaningful research into the correctness of OIDCfed, there has to be a more stable release of the software that has caught up with the newest version of the specification. Not directly to this or future research, a more stable release (e.g. with more stable dependencies), would make this piece of software easier in use, and a more approachable research target. The current lack of documentation combined with the inability to use without the right configuration makes it harder to validate its behaviour.

7.1.2 On OIDCfed Testing

The model, and especially the adapter, have been designed against the working component (the Signing Service). A significant part of the OIDCfed software behaviour, however, is put in the relying party

[1] The concept where a software engineer explains code to an inanimate object such as a rubber duck

(RP), which has to trust (a chain of) Signing Services. The algorithms related to this behaviour has not been tested at this point. Future work would therefore include the validation of the RP, potentially using the model suggested in this work - combined with an adapter that still has to be developed.

Models that have been created or inferred from either the specification or an implementation might be model checked to find potential (rare) faults in the system, or be used as input for the Model Based Testing approach. These models, combined with fuzzing for instance, can verify the full chain as noted above. Fuzzing would serve as useful input to the encoding and cryptography library verification. Model Based fuzzing, or fuzzing as part of Model Based Testing, might find issues with the application with regards to the Signing Service, for instance.

On the specific tests that have been performed in this research, more of the (implied) verification of specific parts of software in the adapter, could be moved to the actual model. What it means to be a correct response from the Signing Service comes with quite a few specifics, that should not be expressed solely in the adapter, but for verification purposes, should be defined in the model. This can be done if there is a clear way of incorporating logic related to encoding and cryptography, or a solution for passing close to all data onto the model should be implemented.

7.1.3 On Testing of OIDCfed Dependencies

This research focused on the testing of that what has been written down in the specification. Part of this, however, relies on external logic, such as signing and JWT libraries. The status of these components with regards to their validation is unknown, and might need further inspection in order to gain full trust in all of the components of this OIDCfed system.

As noted in the related work, there are also model learning strategies that can be applied to the OIDCfed system. These models can be inferred from both static analysis, in which case there is a very clear link to what line of code does what, or can be inferred from behaviour, as has been done with SSH. The resulting models might also have different purposes, and can be used to detect different kinds of problems.

Acknowledgements

This work had not been possible without the help of a number of people. In particular, this practical way of model based testing - which made this work feasible - had not been possible if Axini had not developed their pleasant platform. In particular, Machiel and Peter helped me through my MBT journey without fail. It is their expertise that helped me shape this research, and eventually come to my findings.

My appreciation also goes out to Roland Hedberg, the editor of the OIDCfed specification and implementation - the target of this research. His quick responses and general approachableness has made verifying findings and results a very pleasant experience.

Last but not least, I would like to thank my colleagues at Nikhef's PDP group, Mischa and David, for helping me with any and all knowledge I was missing related to federations, trust, OIDC and the world wide communities around these topics.

Bibliography

[1] N. Sakimura, J. Bradley, M. Jones, B. de Medeiros and C. Mortimore, 'Openid connect core 1.0 incorporating errata set 1', *The OpenID Foundation, specification*, 2014.

[2] S. Cantor, 'Saml version 2.0 errata 05', *OASIS Approved Errata*, 2012.

[3] W. W. Royce, 'Managing the development of large software systems', in *Proceedings of IEEE WESCON 26*, IEEE Computer Society Press, 1970, pp. 1–9.

[4] J. Tretmans, 'Model based testing with labelled transition systems', in *Formal methods and testing*, Springer, 2008, pp. 1–38.

[5] ——, 'Conformance testing with labelled transition systems: Implementation relations and test generation', *Computer networks and ISDN systems*, vol. 29, no. 1, pp. 49–79, 1996.

[6] R. Hedberg, A. A. Solberg, S. Gulliksson, M. B. Jones and J. Bradley, *Openid connect federation 1.0 - draft 06*.

[7] P. Fiterau-Brostean, T. Lenaerts, E. Poll, J. de Ruiter, F. W. Vaandrager and P. Verleg, 'Model learning and model checking of ssh implementations.', in *SPIN*, 2017, pp. 142–151.

[8] B. K. Aichernig, W. Mostowski, M. R. Mousavi, M. Tappler and M. Taromirad, 'Model learning and model-based testing', in *Machine Learning for Dynamic Software Analysis: Potentials and Limits*, Springer, 2018, pp. 74–100.

[9] D. Remenska, 'Bringing model checking closer to practical software engineering', 2016.

[10] W. Mostowski, E. Poll, J. Schmaltz, J. Tretmans and R. Wichers Schreur, 'Model-based testing of electronic passports', 2009.

[11] P. Van de Laar, J. Tretmans and M. Borth, *Situation awareness with systems of systems*. Springer Science & Business Media, 2013.

[12] *Edugain policy framework*.

Glossary

"Federation Operator" "Organisation providing or commissioning the infrastructure for Authentication and Authorization to the members of its Federation.[12]".

"Federation" "Identity federation. An association of organizations that come together to exchange information as appropriate about their users and resources to enable collaborations and transactions.[12]".

"OIDCfed" A specification and implementation related to federation of trust related to OpenID Connect, by placing trust in a party operating a so called trust anchor, maintaining a set of operators that provide digital identities in a trusted manner.[6].

"SURFconext" "The Dutch national identity federation. 'Grouping' identity providers to simplify collaboration. A Federation Operator.".

"Security Assertion Markup Language" The Security Assertion Markup Language is an XML-based, open-standard data format for exchanging authentication and authorisation data between parties, in particular, between an Identity Provider and a Service Provider.[12].

"Signing Service" A piece of software that signs OIDCfed Entity Statements of elements within the OIDCfed environment that it trusts. This means it can operate as a Trust Anchor. Signing Services can be scoped to a certain audience, such as Research and Education, or a specific company..

"Trust Anchor" "A part of OpenID Federation which is primarily used to relay trust to. In the case of the reference implementation this is implemented by a Signing Service.".

Acronyms

AAI Authentication and Authorization Infrastructure.
JWT JSON Web Token.
MBT Model Based Testing.
OIDC OpenID Connect.
OIDCfed OpenID Connect Federations.
OP OpenID Provider.
RP OpenID Relying Party.
SAML Security Assertion Markup Language.
SUT System Under Test.
WLCG Worldwide LHC Computing Grid.

Appendices

7.2 Trust Anchor Test Adapter

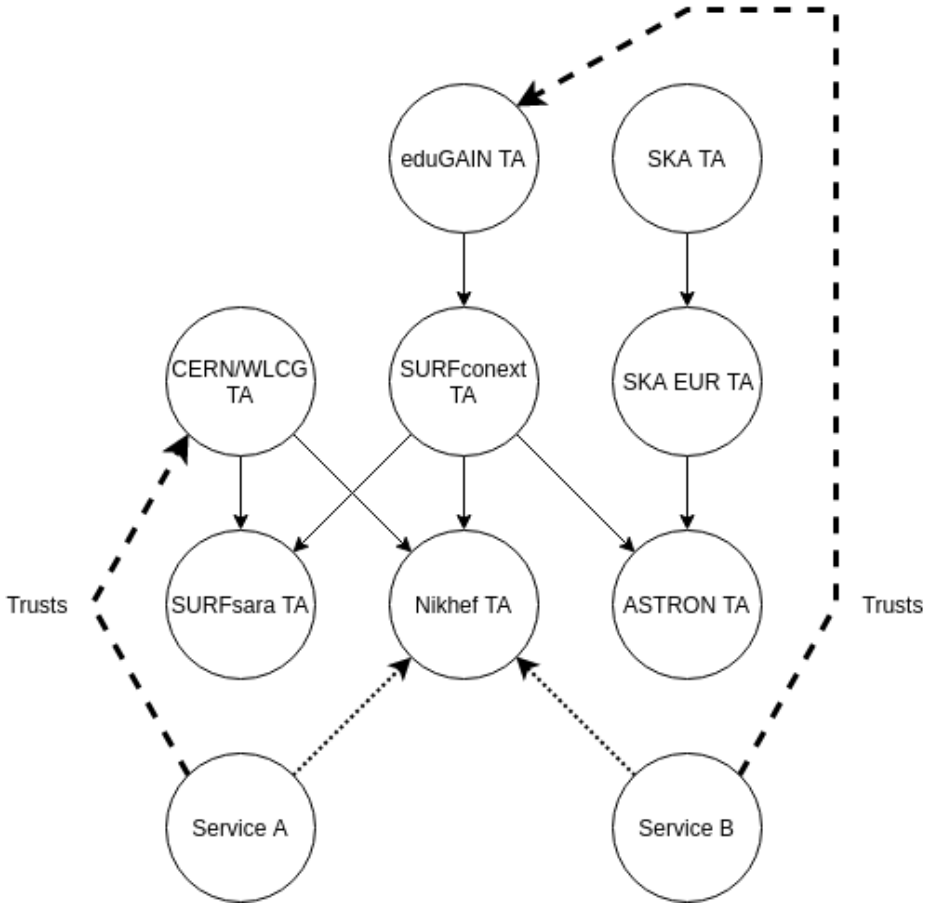
```
require 'cgi'
require 'jwt'
require 'jwt/error'
require 'net/http'

class AdapterHandler < PluginAdapter::HandlerTemplate
  include PluginAdapter::Api::Helpers

  # Processes a stimulus(of a given Label message. This method also
     sets the
  # timestamp and physical label on the Label object. The
     BrokerConnection
  # handles the confirmation to TestManager itself.
  # @param [PluginAdapter::Api::Label] label
  def process_stimulus(label)
    label_name = label.label

    client = Net::HTTP.new("host.domain.tld", 8443)
    client.use_ssl = true
    client.verify_mode = OpenSSL::SSL::VERIFY_NONE

    case label_name
    when 'get_valid_cert'

      fetch_url = '/.well-known/openid-federation?iss=' + CGI::escape(
         @configuration["anchor"])

      if get_parameter(label.parameters, '_withsub', 'boolean')
        fetch_url << '&sub=' + CGI::escape(@configuration["subject"])
      end

      response = client.get(fetch_url)

      token = response.body

      jwt = JWT.decode(token, nil, false).first

      send_label(response('valid', [parameter('_iss', value(":string",
         jwt['iss'])),
                                    parameter('_sub', value(":string",
                                       jwt['sub']))]))

    when 'get_invalid_cert'
      response = client.get('/.well-known/openid-federation')
```

```
      token = response.body

      begin
        JWT.decode token, nil, false
      rescue JWT::DecodeError
        send_response_after('invalid')
      end

    else
      send_error(error("Unsupported Action"))

    end
  end

  def send_response_after(name, seconds = 0)
    Thread.new do
      sleep seconds
      send_label(response(name))
    end
  end

  def get_parameter(params, name, type)
    params.map { |x| [x.name, x.value] }.to_h[name][type]
  end

  # Connects to the SUT and prepares it for testing.
  def connect
    # No op
  end

  require 'byebug'

  # Prepares the SUT for a new test case.
  def reset
    # No op
  end

  # Gracefully closes the connection with the SUT. Called when closing
      the
  # plugin adapter.
  def close
    # No op
  end

  private

  # @return [<PluginAdapter::Api::Label>] The list of labels this
      plugin
  #   adapter supports.
  def supported_labels
    [
      stimulus('get_valid_cert'),
      stimulus('get_invalid_cert'),
      response('valid'),
      response('invalid')
    ]
  end
```

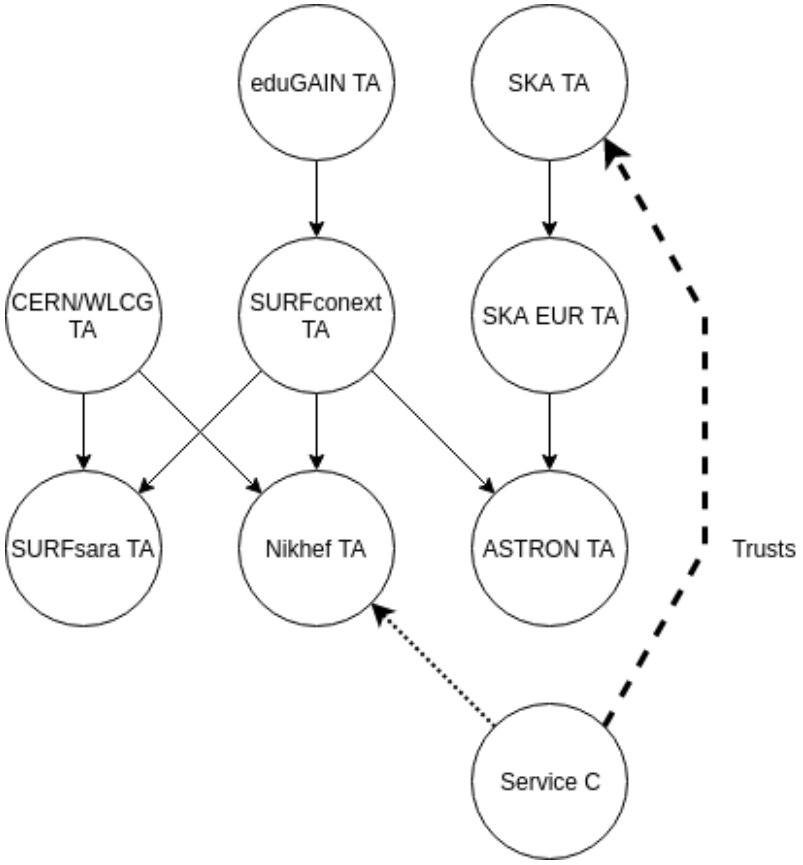
```
  # @return [<PluginAdapter::Api::Configuration::Item>] The list of
  #   configuration items of this plugin adapter.
  def configuration_items
    [
      configuration_item(:name, 'A description', :string, 'default
         value'),
      configuration_item(:anchor, 'Root Trust Anchor', :string, 'https
         ://host.domain.tld:8443/<trust-anchor>'),
      configuration_item(:subject, 'Testing Subject', :string, 'https
         ://host.domain.tld:8888/<OP>'),
    ]
  end
end
```

7.3 A practical federation example

Take the following image: it shows Service A trusting a CERN/WLCG Trust Anchor, while wanting to use an OP that is signed by the Nikhef Trust Anchor. Because the Nikhef TA is signed by the CERN TA which is trusted by Service A, the service knows it can also trust the Nikhef TA, and therefore the signed OP. The same goes for Service B, which trusts the more generic eduGAIN TA. As Nikhef is trusted by SURFconext, and SURFconext is trusted by eduGAIN, the service knows it can also trust the Nikhef signed OP.



Now assume another environment, one where there exists a Service C. This service is aimed at the SKA project which is currently being started. To facilitate this, the SKA could run their own Trust Anchor, incorporating all parties that are involved in the collaboration. Due to an unspecified reason, Service C might want to use an OP that is exclusively signed by the Nikhef TA. Following the trust chains that Nikhef has (SURFconext and CERN/WLCG), we find that no such path leads to the only trust anchor selected. This means that the OP will be rejected.



These two practical examples show various use-cases for federations and how they might look in an OIDCfed environment, but are not in any way an upper limit to the situations that might occur.