Project References

Project Acronym	XIVT		
Project Title	eXcellence In Variant Testing		
Project Number	17039		
Project Start Date	November 1, 2018	Project Duration	36 months
Project Manager	Gunnar Widforss, Bombardier Transportation, Sweden		
Website	https://www.xivt.org/		

Document References

Work package	WP4: Tool Integration and Traceability		
Deliverable	D4.3: Inter-service APIs, format definition		
Deliverable type	Report (R)		
Dissemination level	Public	Date	2020-04-30

Mapped tasks	T4.2: Interface definitions for incorporating WP2 and WP3 outcomes as well as interfaces for partner / use case specific extensions

Executive Summary

This report defines the integration approach between the services / features of the XIVT Variant Testing Toolchain. Based on the High Level Design (D4.2), where tools built by various partners were decomposed into features / services, in this document we define the service endpoints to be tool agnostic.

This report starts by defining the toolchain capabilities, some of which are automatically supported by a tool feature and some others that may require manual triggers (even though with tool support). Toolchain activities, tool features or tool services could be seen as synonyms, depending on the terminology used from the abstraction level of the speech.

A detailed description of those features is in D4.2, in section 4.2, better summarized by the diagram in section 4.1 of the same document. This document D4.3 uses that diagram as reference to define an orchestration sequence (see section 1.2 – Toolchain Activities, page).

The XIVT Toolchain does not require to be used in its entirety per orchestration sequence outlined. The toolchain will provide the ability for user configuration. By selecting and configuring the toolchain for various use cases and features of interest, users can configure the tool for their need.

Each toolchain service is defined with an expected input and output, in an abstract way to guide service level integration of the toolchain. The objective is to standardize the protocol required for the services and features to work together. Input collected from the consortium partners was utilized to standardize the service interface definition, namely ensure that it has all the information needed for that activity.

The interface definition can be seen in two ways:

- First and primary objective is defining the input and output attributes of those features to be tool agnostic and have a standard definition - i.e. a contract between features / service providers (via tools developed) to facilitate integration
- Second, mapping of the input and output attributes to the Technology Stack suggested in D4.2 (as close as possible), keeping it as generic as possible, so that it is feasible to integrate future tools in this toolchain provided they support the interfaces defined.

Some features were supported by multiple tools with varying triggers and responses. In this document we have standardized the definitions to resolve such conflicts to set the direction forward. This means APIs to support the defined protocol and/or some adapters to “translate” the specificities of that tool to the generic interface will be required in the next stages of the project as we work to integrate the various services supported by the tools. This document sets the guideline for expected support from partners to collaborate for the integration activity for D4.4 (Integrated XIVT Toolchain). Any gaps will be addressed during the implementation of the integration of toolchain collaboratively to ensure various services / features supported by the tools are able to cohesively function.

We acknowledge that partners may continue to develop new tools or make changes to existing tools. What is important however is the services and features supported by the tools and they support the interface and protocol defined in this document. All currently available tools were considered in D4.2 and reviewed again in the context of this deliverable for service / feature completeness and to ensure all services and features the toolchain must support have been identified. Protocols defined in this document is the consolidated outcome of that exercise and going forward will be used as the guideline. The division of a specific tool into different services is per the Architecture Direction provided by D4.2 (section 1 and section 3.1), where the XIVT Control UI acts as the facilitator not a central coordinator. This provides flexibility to allow each service the ability to produce and listen to events and decide if an action should be taken or not given the end user needs.

The above paragraph recalls the difference between an Orchestration and a Choreography. The above paragraph directs XIVT toolchain to a Choreography approach (no central coordination), but before achieving it, it is important to start using an Orchestration approach (where there is an "orchestrator" managing the overall service interactions, following a request/response pattern). Chapter 2.1, page , explains in detail why we suggest an Orchestration approach.

To recap. the contents of this report addresses the following:

Toolchain – explains the main sequence of activities and the activities output being tool agnostic

Orchestration – explain the importance of an Orchestration approach and the way to implement such architecture

Services Attributes – details the attributes of the standard protocol, so that the Orchestrator has only one "language", being the responsibility of each tool partner to develop the "translators" to / from their specific services

Tools and Toolchain Mapping – maps each tool to the conceptual activities in the toolchain

Detailed Tools and Attributes Definition – allows each partner to understand where its tool can play a role in the toolchain, and as well, other partners understand the functionality of each tool. This chapter should also describe the adaptors needed by each tool to dialogue with the Orchestrator

Technology Stack – defines the technology stack of the overall integration toolchain

Table of Contents

XIVT Toolchain 8
Introduction 8
Toolchain Activities 11
Toolchain Sequence and Work Packages 16
Toolchain Activities Output 18
Orchestration 21
Description 21
Service Interface Definition 32
High-Level Service Attributes 32
Detailed Service Attributes 34
A01 – Variability Test Case Modelling 35
A02 – Feature Reuse Analysis 35
A03 – Sampling 36
A04 – Instantiation 36
A05 – Test Modelling 37
A06 – Test Reuse Analysis 37
A07 – Test Generation 38
A08 – Test Optimization 39
A09 – Test Adaptation 40
A10 – Test Execution 41
A20 – Mutation Testing 42
A21 – Fault Injection 43
Appendix A: Tools and Toolchain Mapping 46
Introduction 46
Mapping of Tools to Toolchain Activities 46
Appendix B: Detailed Tools and Attributes Definition 48
VarSel: Service for Variant Selection 50
Service for Test Case Adaptation 50
Service for Test Suite Quality Assessment 50

Service for Test Injection and Execution (FBDMutator) 51
Service SEAFOX 52
Service RCM (NALABS) 54
BeVR: Service for requirements-based variability modelling 54
Service for generating human-readable test scripts 54
Service for IIoT component identification 54
Service for risk-based test scoring 55
VARA: Service for requirements similarity-based reuse prediction 55
SaFReL: Service for performance test case generation 57
ReForm: Service for requirement formalization and test optimization 57
ifakVBT: Service for test case generation and variant traceability 58
MTest 59
RELOAD: Service for test load generation 59
TARA 59
DoTA 60
MODICA 61
ARD Test Prioritization Tool 61
MERAN 61
TESTONA 62
InnSpect 62
DeltaFuzzer 64
TV RoboTester 65
Otomat (Testroyer) 66

Appendix C: Technology Stack 67

1 XIVT Toolchain

1.1 Introduction

This chapter recaps the details of the Toolchain concepts so that all partners have a common understanding and, consequently, can ensure the services and features supported by the various tools can be integrated into the toolchain.

To normalize terminology, a “tool” represents an entity that encomposses one or more “features” or “services” required for variant testing. For example, a tool may be called the “awesome tool” but what’s of interest is what this awesome tool does such as “Test Generation” and/or “Test Execution” in the context of variant testing. Therefore, moving forward, reference to integration shall be in the context of services or features.

With that in mind, some important considerations:

- There is one Orchestrator using a standard protocol, which any service must use in order to be triggered by and to respond to the XIVT Control User Interface
- In addition, each service receives and sends information using a standard protocol as defined here

Please note that due to conflict among existing triggers and output formats, it is expected that some tools will require tweaks and updates to APIs and/or need a “driver” (or adaptor) to convert internal data representation to/from the desired standard protocol to facilitate toolchain integration.

Each tool must associate each functionality to one elementary service. Each service implements one Toolchain Activity, as it is represented in the following diagram:

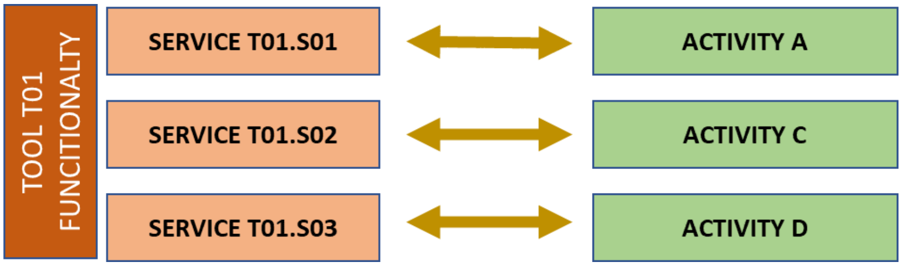


Figure 1.1 – Conceptual association between Tool Services and Toolchain Activities

To recap the High Level Design, there are situations where a single tool supports implements several activities in sequence, and therefore, the owner of Tool T01 (using the example above) needs to treat that functionality as three services (called S01, S02 and S03) to implement three independent toolchain activities by supporting appropriate protocols defined in this document.

If we see this example under the perspective of the toolchain, we have one sequence of activities managed by an Orchestrator to ensure Activity A, B, C, D and E. The Orchestrator received information from UI (User Interface) to know which tool is supposed to be used, and using the example above, we can represent the toolchain as follows:

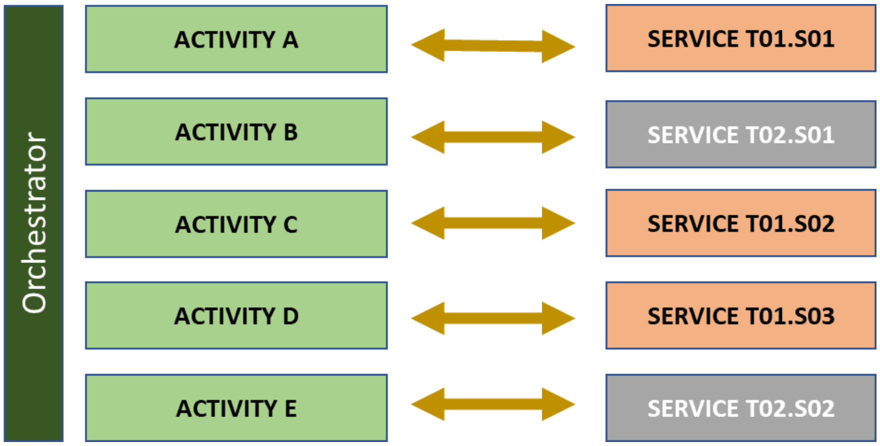


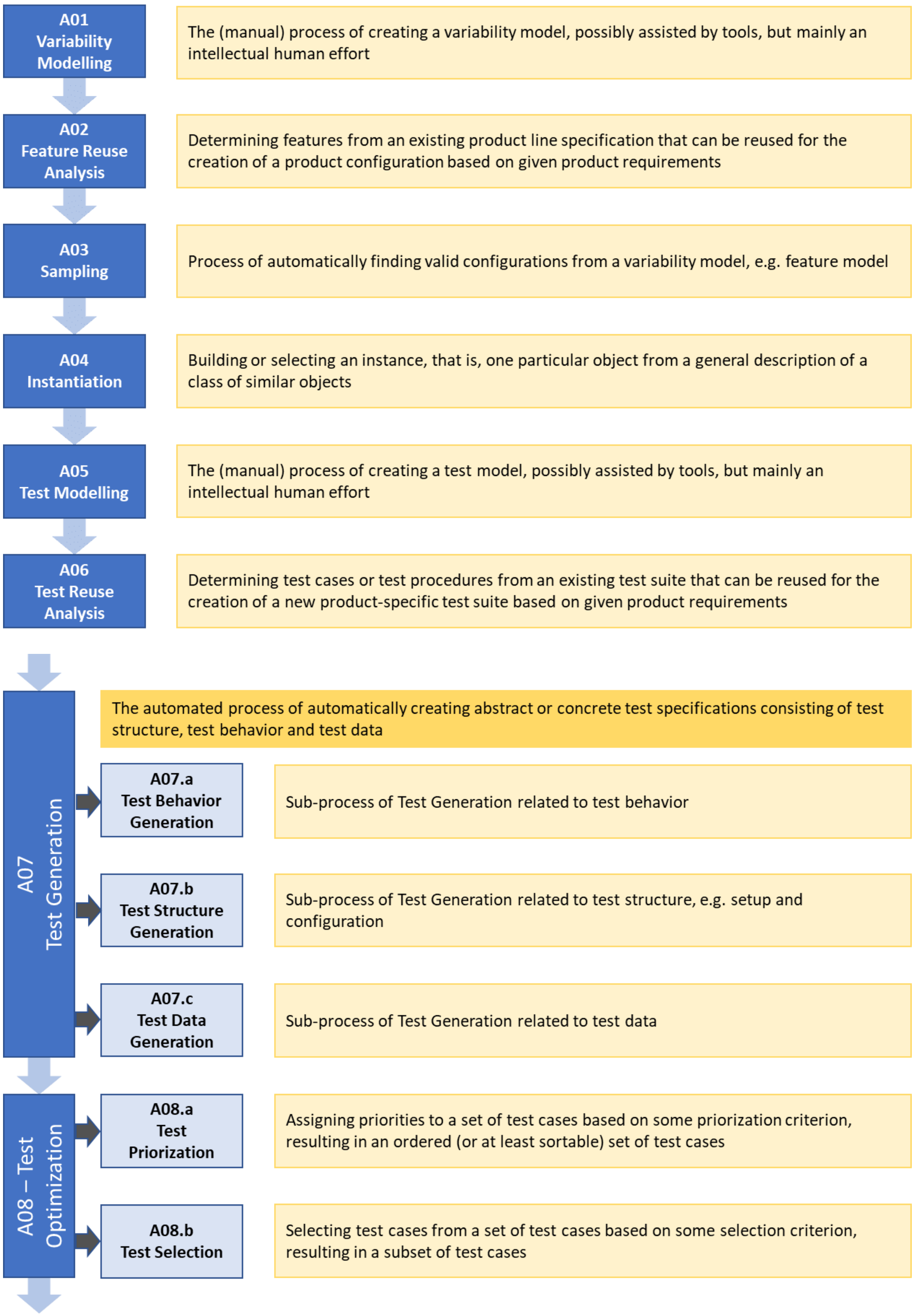
Figure 1.2 – Toolchain Activities Sequence and the associated Tool Services

The Orchestrator uses two tools, named T01 and T02 in this example. The first tool implements Activity A, C and D, while T02 implements Activity B and E. It is this flexibility that we can see in the following sections of this document and, as well, the definition of the abstraction concepts to implement that level of flexibility.

With this explanation it will be easy to figure out that “Toolchain Activity”, “Tool Feature” or “Tool Service” can be seen as synonyms. We will use each terminology depending on the abstraction level of the speech.

The next section explains the main lifecycle / work stream of the toolchain activities – activities 01 until 10. There are also two complementary sets of tools, called “Complementary Tools” and “Analysis Tools”, both under the umbrella of the “Test Suite Quality Assessment” concept. The “Complementary Tools” set includes Mutation Testing and Fault Injection concepts, and they are numbered Activities 20 and 21. The “Analysis Tools” set includes Coverage Analysis and Test Model Checking concepts, and they are numbered Activities 30 and 31.

1.2 Toolchain Activities



A01 Variability Modelling

The (manual) process of creating a variability model, possibly assisted by tools, but mainly an intellectual human effort

A02 Feature Reuse Analysis

Determining features from an existing product line specification that can be reused for the creation of a product configuration based on given product requirements

A03 Sampling

Process of automatically finding valid configurations from a variability model, e.g. feature model

A04 Instantiation

Building or selecting an instance, that is, one particular object from a general description of a class of similar objects

A05 Test Modelling

The (manual) process of creating a test model, possibly assisted by tools, but mainly an intellectual human effort

A06 Test Reuse Analysis

Determining test cases or test procedures from an existing test suite that can be reused for the creation of a new product-specific test suite based on given product requirements

A07 Test Generation

The automated process of automatically creating abstract or concrete test specifications consisting of test structure, test behavior and test data

A07.a Test Behavior Generation

Sub-process of Test Generation related to test behavior

A07.b Test Structure Generation

Sub-process of Test Generation related to test structure, e.g. setup and configuration

A07.c Test Data Generation

Sub-process of Test Generation related to test data

A08 – Test Optimization

A08.a Test Priorization

Assigning priorities to a set of test cases based on some priorization criterion, resulting in an ordered (or at least sortable) set of test cases

A08.b Test Selection

Selecting test cases from a set of test cases based on some selection criterion, resulting in a subset of test cases

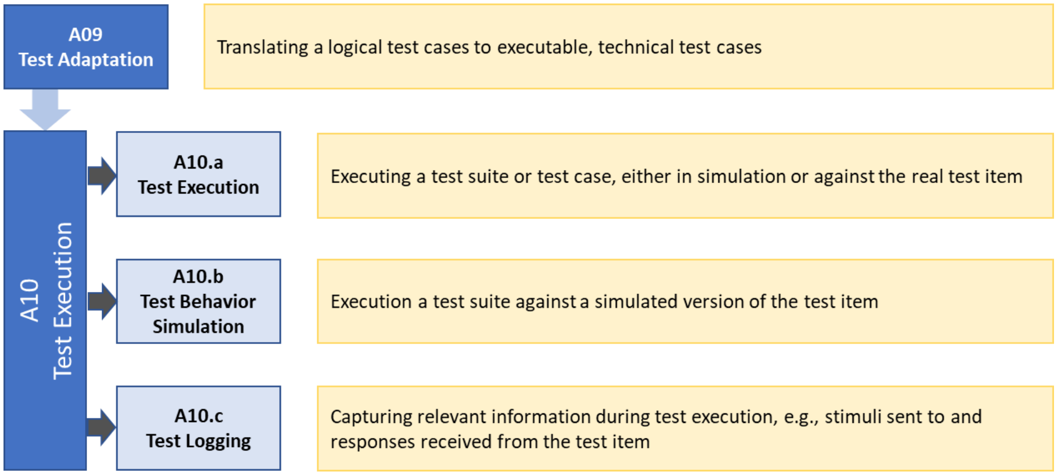

A09
Test Adaptation
Translating a logical test cases to executable, technical test cases
A10
Test Execution
A10.a
Test Execution
Executing a test suite or test case, either in simulation or against the real test item
A10.b
Test Behavior
Simulation
Execution a test suite against a simulated version of the test item
A10.c
Test Logging
Capturing relevant information during test execution, e.g., stimuli sent to and responses received from the test item

Figure 1.3 – Toolchain Activities Sequence

This sequence of Activities, from A01 until A10, represent the main purpose of the Toolchain. The name of each activity is associated with the concepts already explained in D4.2. In this document, we organize those concepts into a logical sequence of activities as applicable for variant testing and further define details behind each activity.

Although in the diagram above it is explained each activity with a description in the right side, it is important to highlight some clarifications:

1. **Variability Modeling** – this activity is mainly an intellectual human effort and it is supposed to provide, as output, a model description of business variability. Although it is a human effort activity, it is supposed to be supported by a tool, namely a tool that reads textual specifications and, with NLP (Natural Language Processing) provides the desired model description that will be used as input data for the second activity.

2. **Feature Reuse Analysis** – this activity uses the model description of A01 to identify similarities between variants and cluster them.

3. **Sampling** – the output of A02 is used by this activity to identify if they are invalid variants and remove them from the specification model. For example, if a configuration option implies that another configuration doesn't make sense, it is useless to define a test case with the combination of those two functionalities. Therefore, we see this activity as the point where we move from a "Variability Model Description" to a "Product Model Description".

4. **Instantiation** – this activity uses the "Product Model Description" from A03 where there are generic classes (representing conceptual variants) and instantiates each class in a specific way, so that we have the source material to start the Test Modelling.

   Activities A02, A03 and A04 comprise one of the technological innovations of XIVT Project, based on knowledge-based requirements analysis and selection, extracting features and requirements using machine learning techniques and map-reduced algorithms, with the associated identification of features and priority ranking for testing.

5. **Test Modelling** – this activity uses the Product Model Description from A04 to create a Test Model. The Test Model will derive in a set of Test Conditions, supported by the modelling data, with possible reusage combinations between those test conditions (see activity A06), but it is still missing detailed information for the testing steps of each test case (which is addressed by activity A07).

6. **Test Reuse Analysis** – this activity uses the "Test Model Description" from A05 and identifies similarities and clusters them, so that we have an unique set of Test Conditions.

7. **Test Generation** – this activity uses the data of A06 to detail each test case into a test specification. Each test has the description of the preconditions to execute the test (test structure), the test data needed and the test steps (test behavior) needed to accomplish the software validation of that test case. We can see this

activity as the aggregation of 3 sub-processes as represented in the above diagram.

8. **Test Optimization** – with the output of A07, we could imagine that we have all required information to start test execution. However, it is not the case, since XIVT project intends to extend the optimization as much as possible. Therefore, although we have at A02 and A06 some optimization by reusability, here the optimization is by reducing the test case list. With the detailed information provided by A07, we can use some parameters to score each test case by priority (using best practices like Risk Based Testing). With that scoring (activity A08.a), activity A08.b can discard some test cases with very low priority, reducing the test case list without increasing the business risk (or with a marginal and insignificant increase).
   NOTE: the concept of Test Optimization could be seen in two different ways inside XIVT project (see section 1.3 – Toolchain Sequence and Work Packages, page for details)

9. **Test Adaptation** – with the output of A08, with the detail of each test case description, we can move from "logical test cases" to "executable test cases", or if we want, from "abstract test cases" to "concrete test cases". This activity A09 can also be seen as a compiler, that translates human readable statements into executable instructions for an engine to automatically do the testing itself.
   This activity is preceding A10 activity if we are under the context of a "compiler". If the test execution engine interprets a meta-language in real-time, we can see A09 and A10 as only one activity for each test step because the translation of "logical" to "executable" instructions will be done during test execution.

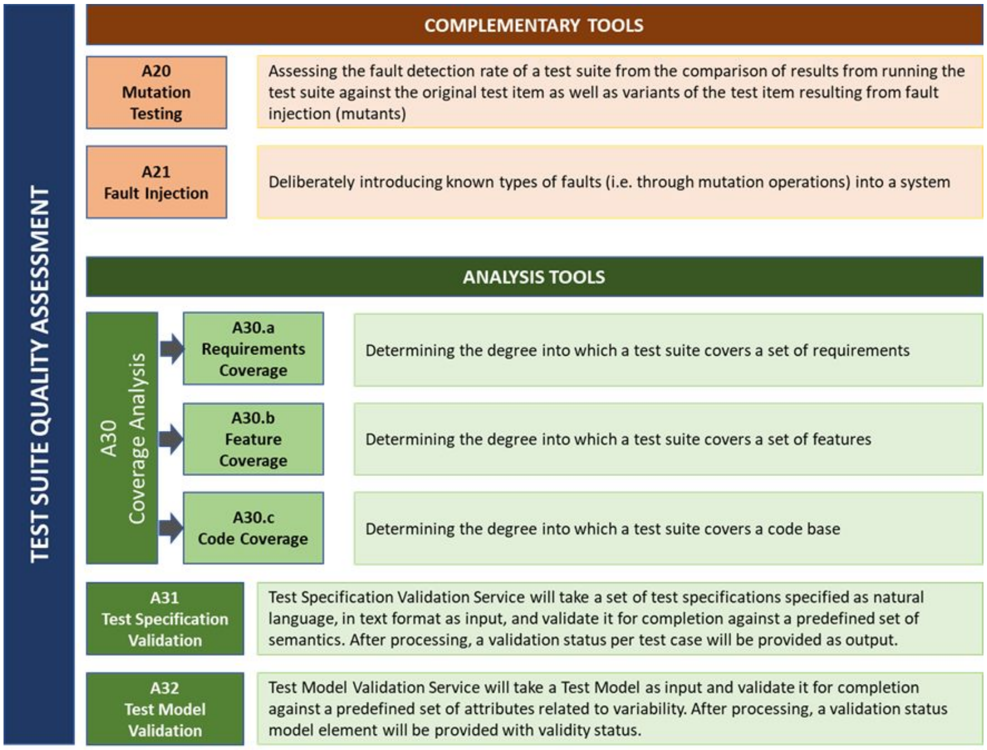
10. **Test Execution** – this activity can also be seen as the aggregation of 3 sub-processes as represented in the above diagram. One is the execution itself; another is the simulation of one test item (for example, a hardware component or a service from a 3rd party and it is not available yet); and the third one is the execution logs to store evidences of the test itself for analysis purposes and for defect management (if the test detects a defect).

Complementary to the above sequence, there is also a “Test Suite Quality Assessment”, which is used to validate the accuracy of the toolchain itself. The diagram below explains those activities where we can see:

- **Complementary Tools** – those activities A20 and A21 complements A10 since it introduces mutations and fault injection to validate if the toolchain can detect those mutations, and therefore, provide us the desired level of confidence

- **Analysis Tools** – These activities A30 and A31 allow us to identify if the toolchain is providing the desired level of coverage, and therefore, provide a quality metric of the testing effectiveness.

Activity A31 also represents a model checking procedure for selecting applicable test cases for specific products in the regression test of product lines. The conceptual approach behind it is the certification of software on the basis of the product family, which allows to maintain certificates for parts that have been previously validated.



A20 Mutation Testing: Assessing the fault detection rate of a test suite from the comparison of results from running the test suite against the original test item as well as variants of the test item resulting from fault injection (mutants)

A21 Fault Injection: Deliberately introducing known types of faults (i.e. through mutation operations) into a system

A30.a Requirements Coverage: Determining the degree into which a test suite covers a set of requirements

A30.b Feature Coverage: Determining the degree into which a test suite covers a set of features

A30.c Code Coverage: Determining the degree into which a test suite covers a code base

A31 Test Specification Validation: Test Specification Validation Service will take a set of test specifications specified as natural language, in text format as input, and validate it for completion against a predefined set of semantics. After processing, a validation status per test case will be provided as output.

A32 Test Model Validation: Test Model Validation Service will take a Test Model as input and validate it for completion against a predefined set of attributes related to variability. After processing, a validation status model element will be provided with validity status.

Figure 1.4 – Test Suite Quality Assessment

1.3 Toolchain Sequence and Work Packages

This section is a pertinent one because it maps the conceptual diagrams described in FPP Annex with the toolchain activities. This is even more important because some terminology used in FPP Annex uses the same word as it is used in toolchain but under a different perspective.

Recovering the picture at the end of FPP Annex, section 2.1.1, we have the following diagram:

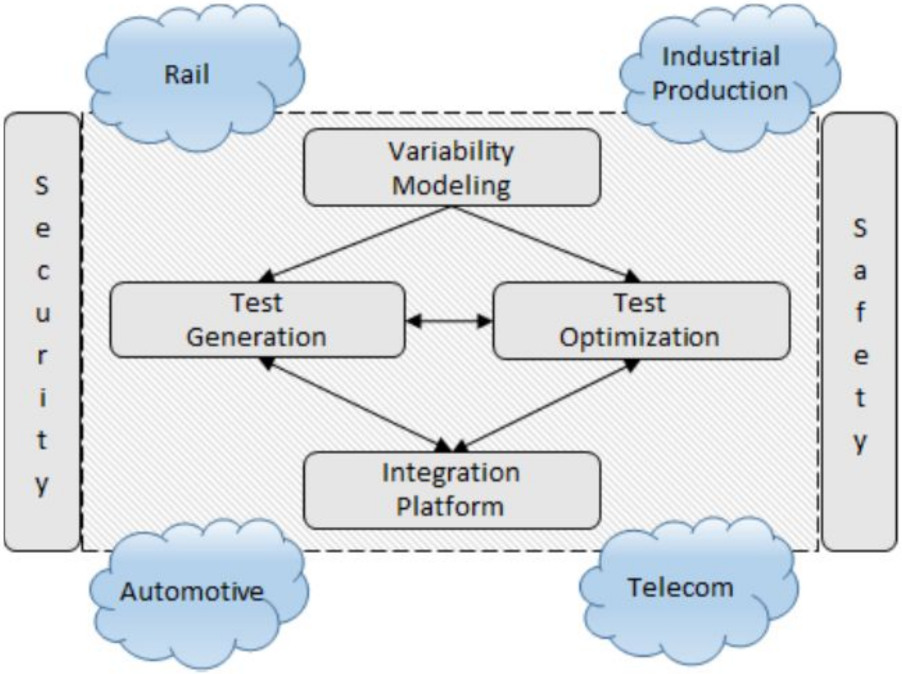


Figure 1.5 – XIVT innovation fields in variant testing, across industries

The Variability Modelling concept is represented by Activities A01 to A06, Test Generation by A07 and Test Optimization by A08. Activities A09 and A10 can be seen as the materialization of the end result of the Integration Platform.

Using the Work Package view, in special the four technical work packages (WP1, WP2, WP3, WP4), we can see in the following diagram the names of the WPs:

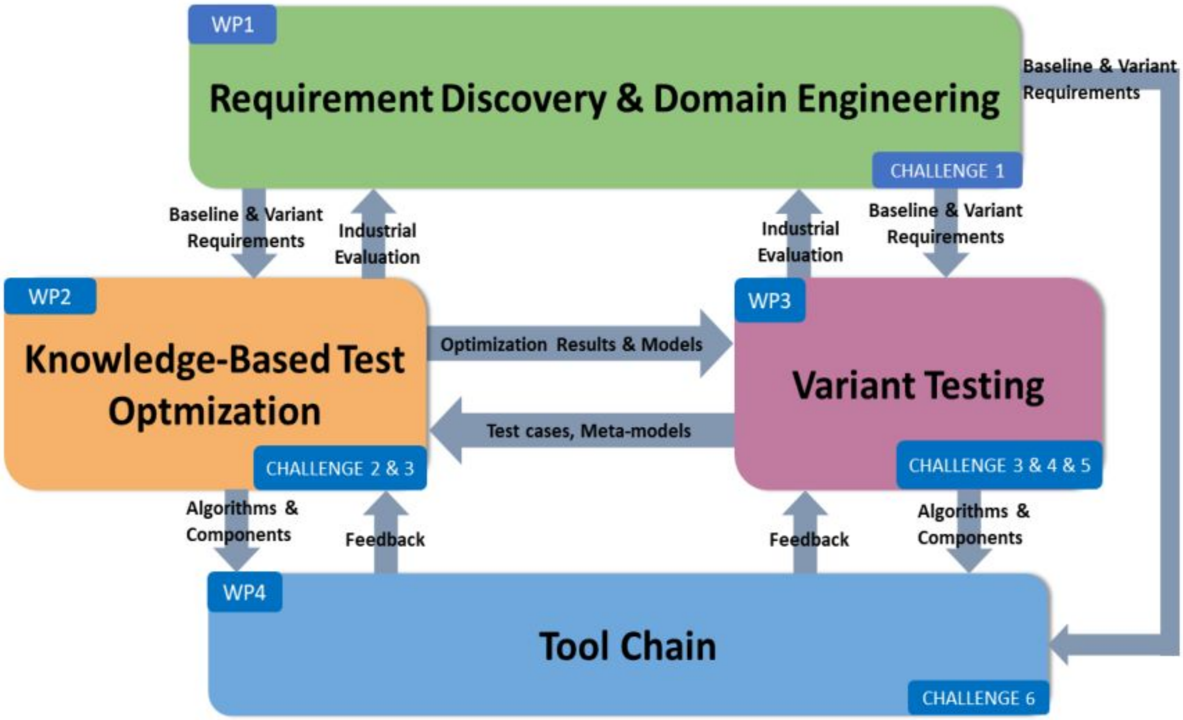


Figure 1.6 – Work Packages diagram

Starting with WP2, named “Knowledge-Based Test Optimization”, it is important to clarify that this optimization is over requirements and abstract test cases.

WP2 addresses Challenge 2 and 3, which are:

- CHALLENGE-2: Prioritization and selection of requirements for testing using knowledge-based techniques
- CHALLENGE-3: Generating abstract test cases using abstract model

In contrast, toolchain activity A08, also named “Test Optimization”, addresses a concrete Test Case List, supported by Test Procedures where it is explained the Test Behavior (the output of Activity A07 – Test Generation).

Looking now at WP3, we see the name “Variant Testing”. This WP3 addresses Challenge 3, 4 and 5:

- CHALLENGE-3: Generating abstract test cases using abstract model
- CHALLENGE-4: Concretizing abstract test cases and assessing them in product
- CHALLENGE-5 Improving software security based on test cases results

As we can see, the “Variability” terminology at Toolchain activities is associated with the modelling parts (activities A01 to A06) and “Variant Testing” Work Package is more associated with A07, A08, A20 and A21.

This apparent inconsistency does not create a real problem because all XIVT Project is about variability and testing. The difference is the level of abstraction of the speech, and this section was written just to avoid some confusion and turn all concepts clear.

In summary, we have WP2 describing “optimization” followed by WP3 describing “variant” testing. Toolchain activity describes “variant” modelling followed by “optimization” at later stages of the activities chain. Of course, at the modelling stage, trying to reduce the number of test cases is (obviously) an optimization, but at toolchain this term is addressed under “Test Analysis and Design” scope and not modelling scope.

1.4 Toolchain Activities Output

Under the same desire to avoid confusion, it is important to use the same terminology when we are talking about input and output data. The terminology proposed is not a finished work, since we need to get feedback from all tool partners, and it could make sense to change a term to a different word to keep more consensus inside XIVT consortium.

Therefore, this section explains the output of each Toolchain Activity to keep all people aligned and everyone understands exactly what the objective of each output is. After this explanation, it will also highlight some incongruencies between the Toolchain Activities and the mapping and tool classification described at D4.2. As soon as those incongruencies were solved (at D4.2 or at this D4.3 document), these highlights should be removed from this document. For easier identification, they are put inside a gray box because they are collateral information and not the core information of this document.

Activity	Activity Name	Output ID	Output Description
A01	Variability Modelling	OUT.01	Variability Model Description
A02	Feature Reuse Analysis	OUT.02	Variability Model Description (with Similarity Clustering)
A03	Sampling	OUT.03	Product Model Description (with generic classes)
A04	Instantiation	OUT.04	Product Model Description (with classes instantiated)
A05	Test Modelling	OUT.05	Test Model Description
A06	Test Reuse Analysis	OUT.06	Test Model / Test Conditions (after Test Reuse Analysis)
A07	Test Generation	OUT.07	Test Behavior Description with preconditions and test data (Logical Test Case List)
A08.a	Test Prioritization	OUT.08.a	Logical Test Case List (with priority score)
A08.b	Test Selection	OUT.08.b	Logical Test Case List (test subset)
A09	Test Adaptation	OUT.09	Executable Test Cases (Concrete Test Scripts)

After having the executable test scripts, the tool engine does the job of automatic test execution (A10) complemented by Mutation Testing (A20) and Fault Injection (A21).

Mutation Testing is outside the toolchain main sequence because it can be seen as pre-execution or post-execution activity.

If A20 receives a list of test cases and mutation operators, the end results will be a new test case list generated before starting execution. Another situation is doing fault injection at SUT (System Under test) and analyzing, after execution, the defects automatically detected. See diagram below.

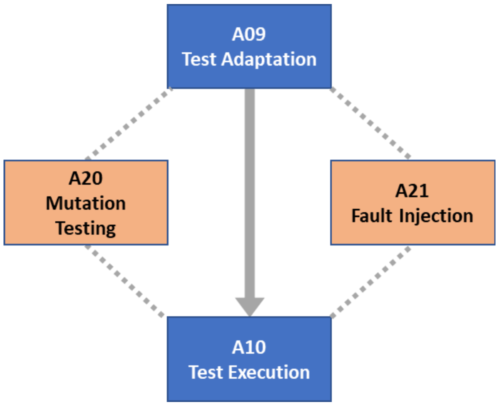


Figure 1.7 – Mutation Testing and Fault Injection as complement of main activity sequence

2 Orchestration

2.1 Description

In order to cohesively integrate various services and features to achieve a toolchain, it is necessary the features and services supported by various tools communicate correctly and synchronously according to a standard protocol. This will provide the ability to extend the toolchain in the future with additional tools that provided services and features beneficial for variant testing.

Tools at the beginning of the project were developed by different partners without coordination and alignment on internal behaviors and architecture during their development. Also, those tools were purpose built possibly targeting different domains. The objective of D4.2 and D4.3 is to decompose the features and services supported by different tools and standardize on interfaces and protocols so that they can be cohesively integrated. Please note however, the end user will have the ability to configure the toolchain as they see fit. This will allow features and services that are complementary to be leveraged optimally or in case of disparate features and services, utilize them on a standalone basis if appropriate.

From end to end orchestration perspective, the of services / features can be shown in the figure below.

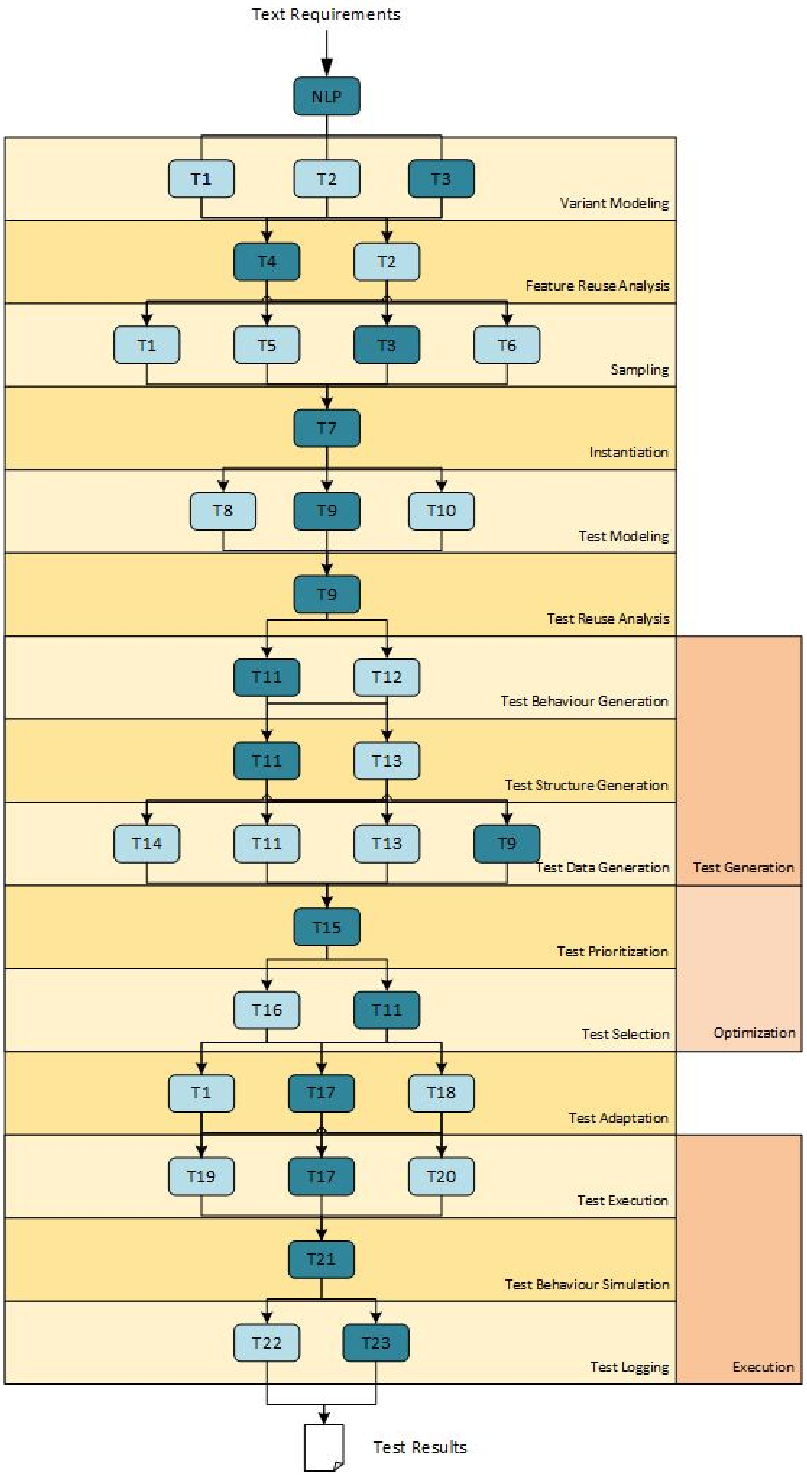

Text Requirements
NLP
T1
T2
T3
Variant Modeling
T4
T2
Feature Reuse Analysis
T1
T5
T3
T6
Sampling
T7
Instantiation
T8
T9
T10
Test Modeling
T9
Test Reuse Analysis
T11
T12
Test Behaviour Generation
T11
T13
Test Structure Generation
T14
T11
T13
T9
Test Data Generation
Test Generation
T15
Test Prioritization
T16
T11
Test Selection
Optimization
T1
T17
T18
Test Adaptation
T19
T17
T20
Test Execution
T21
Test Behaviour Simulation
T22
T23
Test Logging
Execution
Test Results

Figure 2.1 – Toolchain supported by a pipeline of tools

There are two main approaches in order to create the desired synchronization across all different services / features that are part of the toolchain. Those approaches are Orchestration and Choreography.

An Orchestration approach requires one or more software components that are responsible for synchronizing all different applications during the execution of the desired process. Let's use the example illustrated above to understand how an orchestration approach could be used.

As we can see in the previous diagram, the user wanted to execute all the features (testing layers) using a specific tool supporting each feature / layer, and by providing the text requirements and the configuring data to the orchestrator.

The process starts with the user input into the orchestrator. By following the user configurations, the orchestrator distributes the work as it was required. Starting by the "Variant Modeling Layer", the orchestrator selects the appropriate tool that supports a particular service that the user has chosen for that layer – tool T3 in the diagram above.

Once the execution of the service from the first layer is finished, either a message is sent to the orchestrator to fetch the output of the service that have just finished its work (asynchronous case) or the result is sent directly as a response of the strait trigger of the orchestrator (synchronous case). When the orchestrator receives the response of the service that was intended to execute at that specific time, that output is directed via the standard input for the next service and sent to the service / feature of the application that the user has configured for execution in that next layer. The process repeats until the end of the chain is reached or until the user has configured so. The following image provides a visual representation of the orchestration architecture.

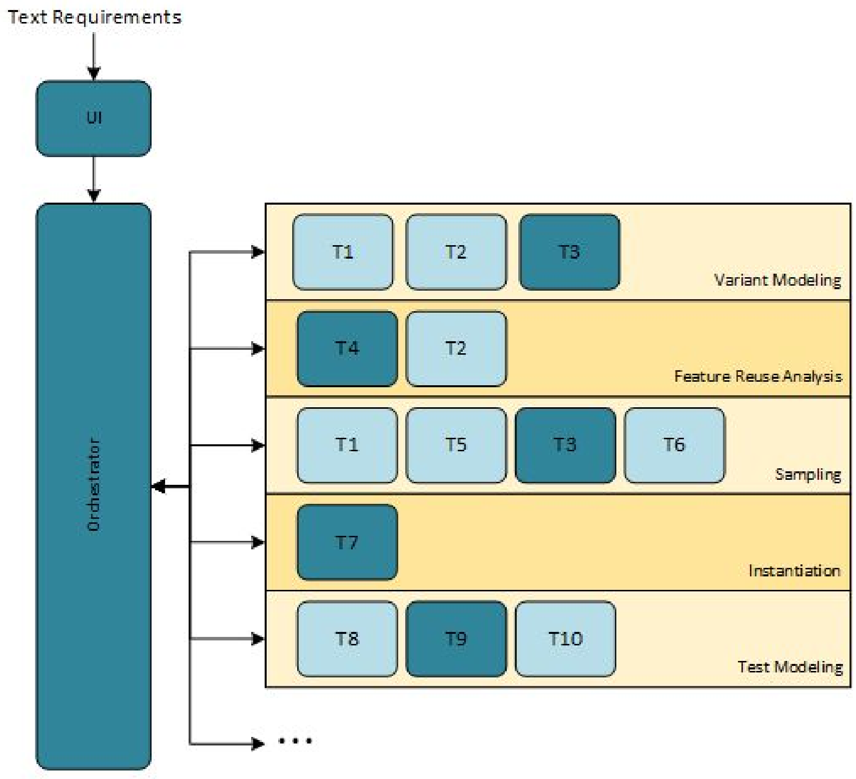


Figure 2.2 – Orchestration Architecture

In the case of a Choreography, the different applications synchronize between themselves without any help of an external application. In a choreography scenario each application knows what and when to send information to the next layer. Commonly publisher - subscriber design pattern is used where services publish output to different topics and services that subscribe to different topics would be monitoring data being posted to topics of interest and start processing whenever there is new information.

Using again the same example used above, for describing orchestration, we have a user that wants to execute all layers using different applications on each different layer. The user provides again the text requirements but this time, configurations are sent among the text requirements. Once in choreography there is no central set of control services to synchronize all different components, the different services that must get the information and which tool supporting the service in that layer should be called next.

Also, each service must support the output as defined in this document so that it can become input of the next service in the toolchain. In such a case the user simply defines the starting point. After sending the initial request to the first service in the toolchain, execution will seamlessly continue till an end state is reached. The image below will represent visually how a choreography architecture would behave.

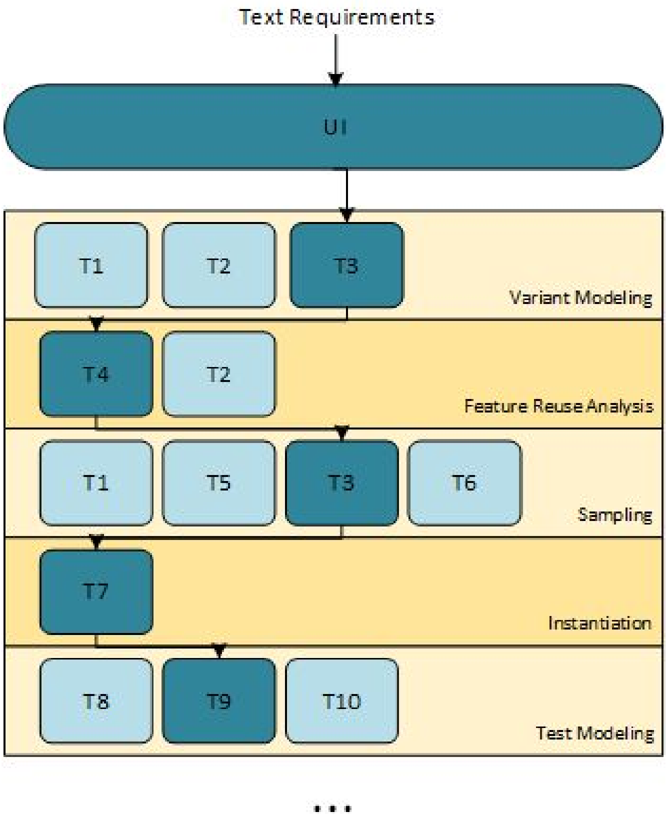


Figure 2.3 – Choreography Architecture

Both architectural patterns have their pros and cons. What is really important is to weigh both pros and cons of both approaches for the specific case which they are going to be applied to.

Now, that both architectural patterns have been explained, let's look a little more into our specific case and try to understand which pattern would fit better into the toolchain. In our toolchain there are some use cases that we must have in consideration. The use case mentioned in the start of this chapter is possibly the simplest one, where a user that wants to use the toolchain chose it from top to bottom with no gaps between lanes. In such cases, both architectural patterns can be applied with no big concerns. Orchestration might have better maintainability, once it is easier to modify all toolchain sequences if one tool changes the connection interface. Choreography on the other hand, might present better performance once the roundtrips to and from the orchestrator does not exist.

Also, the orchestration pattern would require the development of an additional application. However, the choreography pattern would require an extra effort of all partners to incorporate into their tools the ability of converting payloads for different applications and a configuration model that would keep the chain active until further notice from the user.

Those are some of the trade-offs when opting between those two architectures in generic scenarios, but let's take a deeper look into more specific requirements.

As an example, imagine a requirement where the user should be able to execute only specific layers of the toolchain, within certain constraints. The user should not be able to execute layers in a reverse way, and some of the layers might not be compatible with executing one after another, because of lack of information from one lane to another.

Those restrictions can be managed by disallowing the user to select them in the configurations. This freedom provided by the flexibility of the toolchain comes in hand with an increase of its complexity. As so, the toolchain may be used differently than what was represented in the first picture (Figure 2.1, page). The user might choose to execute only a subset of the existing layers, as shown in the following example (layers with no service support in dark color means a jump from the previous to the next level). The sequence might jump levels, start in a middle level or end in a middle level.

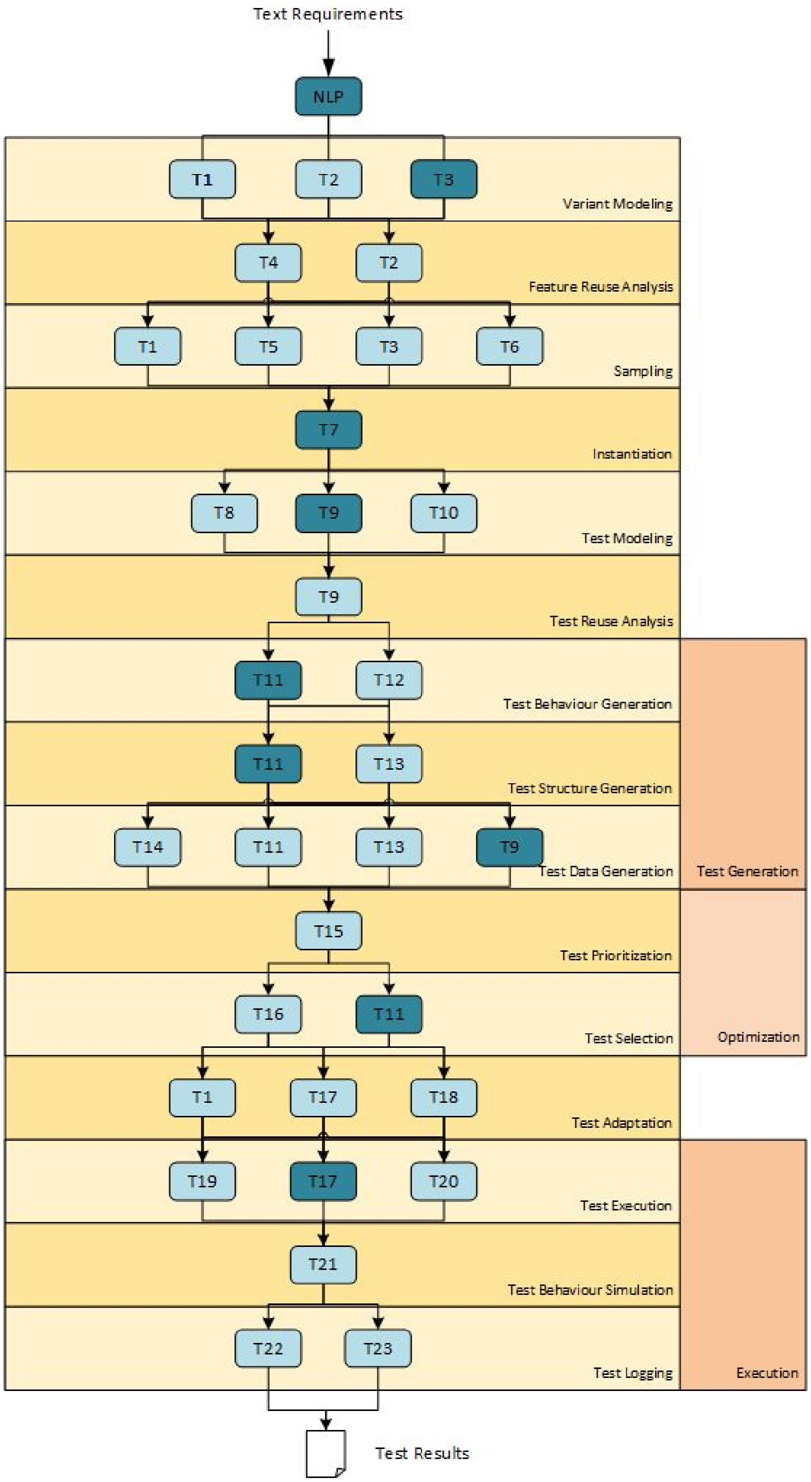

Text Requirements
NLP
T1
T2
T3
Variant Modeling
T4
T2
Feature Reuse Analysis
T1
T5
T3
T6
Sampling
T7
Instantiation
T8
T9
T10
Test Modeling
T9
Test Reuse Analysis
T11
T12
Test Behaviour Generation
T11
T13
Test Structure Generation
T14
T11
T13
T9
Test Data Generation
Test Generation
T15
Test Prioritization
T16
T11
Test Selection
Optimization
T1
T17
T18
Test Adaptation
T19
T17
T20
Test Execution
T21
Test Behaviour Simulation
T22
T23
Test Logging
Execution
Test Results

Figure 2.4 - Toolchain example where some layers are not executed

In this example we can see that the user chooses not to execute some of the existing layers. Opting for an orchestration architecture allows having an application managing the flow which simplifies the implementation pattern. This way the orchestrator could convert the received output to any input payload required by any compatible application at the layers' below.

However, in a choreography scenario, applications should have to be able to send the request for the next tool which might not be linear. In a choreography if an application jumps a layer, such application must produce the input of the layer it is going to send it to. If we analyze Figure 2.4 above, we can see that one application might have to produce n outputs. Those n outputs are not only the inputs of the layer just below but also from all lower layers to which it is compatible to call to.

As an example, if we look at T9 in the Test Reuse Analysis it should not only produce the input for T11 and T12 in the Test Behavior Generation layer but also for any other tool below that layer that is compatible with such a step of the toolchain.

Other approaches might be considered when looking at a choreography architecture. If the service produces a standard output that fits the next layer input, we could develop applications that convert those standard outputs to other layers below using default values. Let's take the same example. The T9 application in the Test Reuse Analysis now needs to call the T11 in the Test Structure Generation, instead of calling it directly it will call an application that belongs to Test Behavior Generation to make the translation of the output from T9 to the input of T11, 2 layers' bellow.

This pattern allows us to simplify each independent tool, despite increasing the workload on standardizing interfaces or adapters per service / layers. If we analyze carefully, this is the job of the orchestrator in an orchestration architecture. Imagine that the user now wants to stop the toolchain execution before the last layer, as shown in the next diagram.

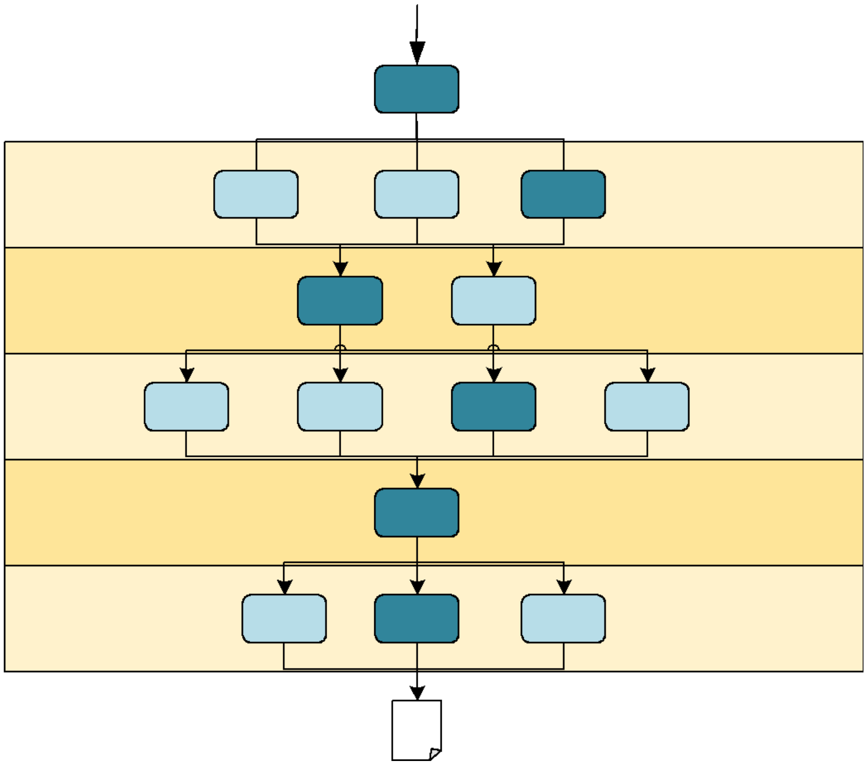


Figure 2.5 – Toolchain example that ends in a middle level

In this scenario the toolchain has to know when the execution should finish. Here, in an orchestration scenario the implementation makes it pretty easy to stop the sequence based on management of the orchestrator. Since the orchestrator receives a configuration parameter to know where it should end, it is easy to implement a decision point that after the orchestrator calls the last service, it should return the result instead of calling another service.

However, when we put this use case on a choreography architecture things get a little bit messier. For the toolchain to stop its execution at a specific point, each service should know if there is a final node or not. Remember that services only get information from the system that called them, and thus, the information about which service should stop the execution has to be shared across multiple services until it reaches the destination service. This would require the implementation of functionality inside the different services that would make those services system dependent. Like so, every service should behave as a node in a graph, knowing exactly to whom it can send the information and if that will be the last system processing the payload. By looking at this use case it is fair to say that an orchestration implementation would fit much better than a choreography pattern.

Now let's take a look into other possible use cases. The user should also be free to start the execution of the toolchain where he would most like. As so the following diagram describes that scenario.

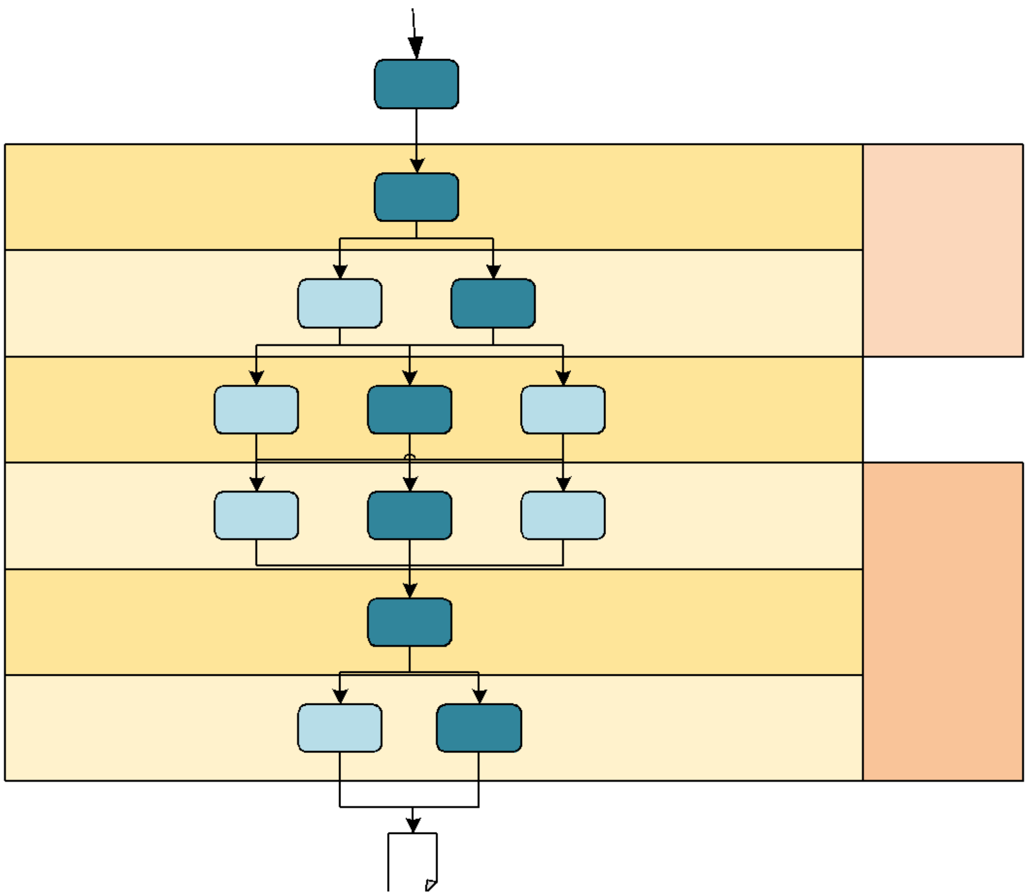


Figure 2.6 – Toolchain example that starts in a middle level

In this situation the UI should know where to send the first request. With some configuration both architectures can be applied with no big deal. In an orchestration architecture, the orchestrator should receive as configuration which should be the first service of the toolchain. On the other hand, in a choreography scenario the UI should send the request directly to the first service of the stack, and so, the UI should have the capability of sending the first request to any possible tool.

Now that we have looked in some of the use cases, it can be seen that even though both patterns have their pros and cons, the orchestration would fit better in the specific scenario.

The major advantages of such architecture are the maintainability of the overall toolchain and the decrease of the workload required to implement such a solution. When looking at maintainability, let’s imagine that a partner wants to add a new service to the toolchain.

If we have an orchestration, the pointer to that service should be managed only on the orchestration side, while if we have a choreography pattern, we would have to change every compatible service and supporting tools to point to the new service / tool. This can be manageable when there are only a handful of services but if that number starts to increase it will become harder to manage as well as more error prone.

Regarding the workload, it can be said that in a choreography scenario each partner would have to support the standard interfaces or develop adaptors for each compatible service. In the case of layer jumps it should also be developed an application that would convert the payloads with some default values, so that the output is compatible with the input of the target service. Even though, in an orchestration architecture most of those things have also to be implemented, with some techniques we can decrease significantly the required work as explained in the sections below (event queues can be utilized).

With the above, interaction among services supported by various tools of the XIVT Toolchain will follow the Orchestrator design pattern as it is more appropriate for a project like XIVT that involves multiple partners providing a variety of services / features via tools.

3 Service Interface Definition

As rationalized in sections 1 and 2, the XIVT Toolchain will be built by integrating a set of services / features required for variant testing using the orchestrator pattern. This section outlines standard interface(s) and protocol(s) required to be supported by each service. Existing inputs and outputs were considered, conflicts were resolved with future maintenance, extensibility and longer term commercial viability in mind.

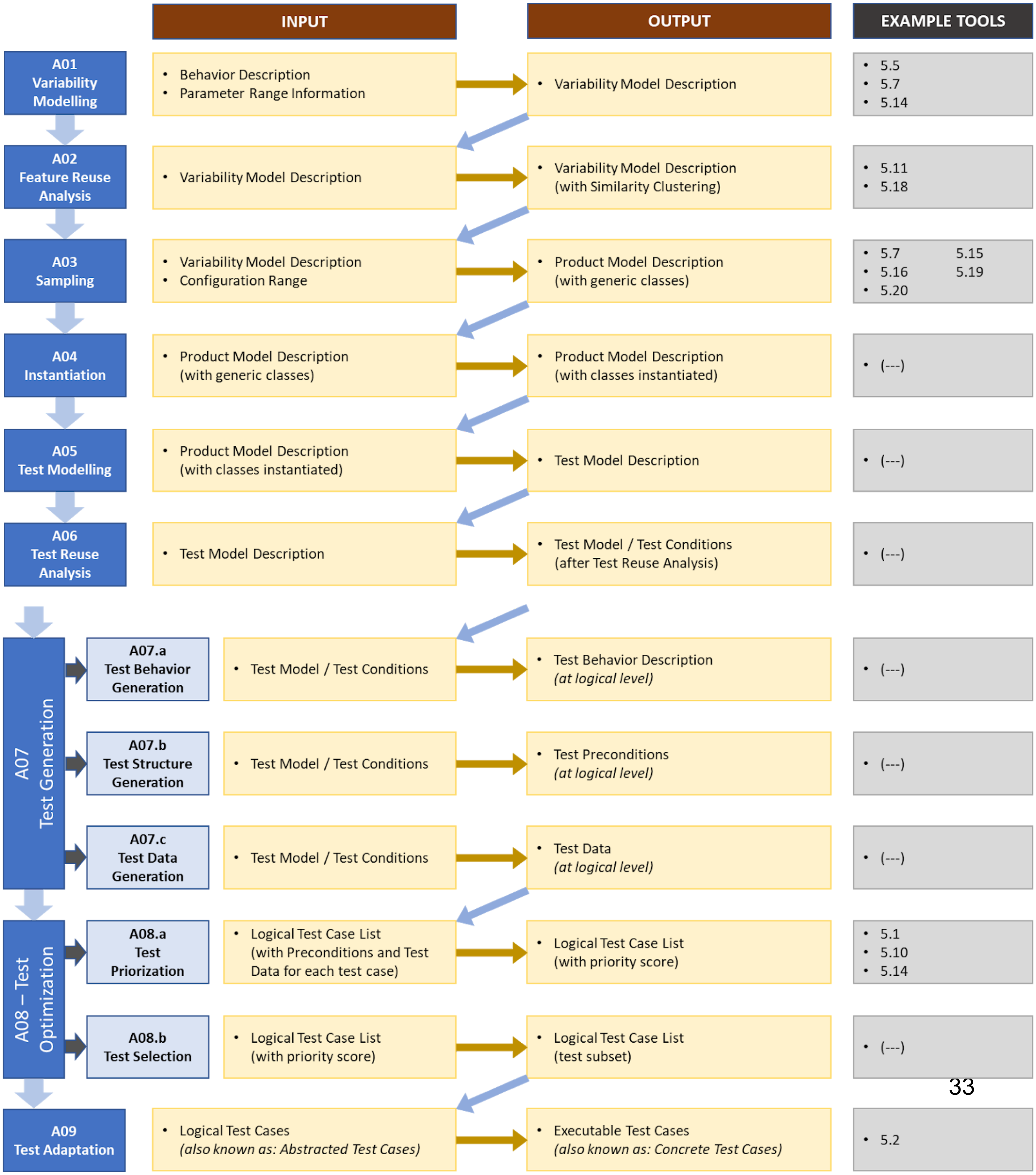
Services (also referred to as features as well as layers throughout the document) are tool agnostic. We have done the exercise, for each service / toolchain activity by asking and answering the questions: what are the needed input attributes for the (abstracted) services to produce the desired output and what the output format should be to allow for integration going forward.

The desired high level inputs/outputs were summarized in section 1.4. In this section, input from partners were gathered and consolidated in order to come to a standard interface definition for the core services part of the XIVT Toolchain. We start by outlining the high level input / output attributes to make it clear to partners why such standardization is required then elaborate further with details for the various tool chain services to act as the baseline for integration.

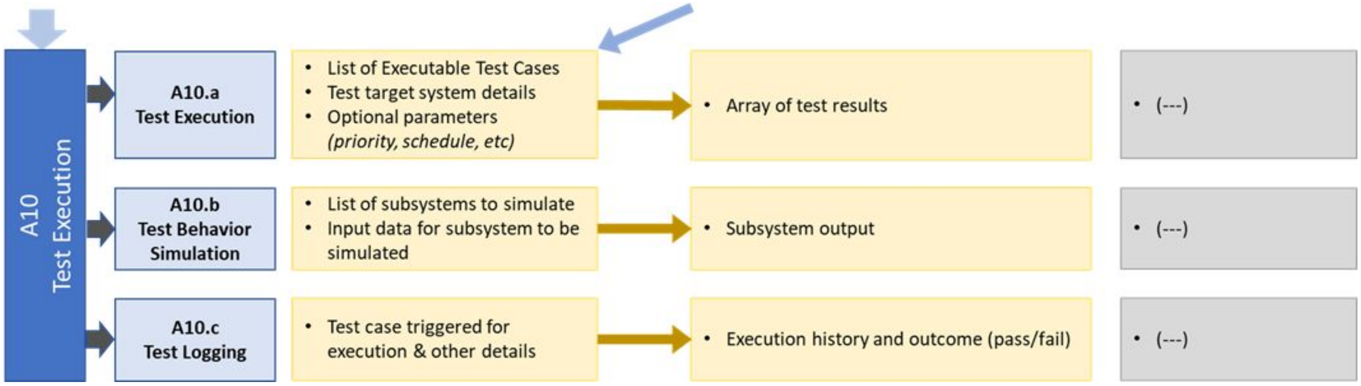
The Orchestrator pattern will use the standard service definition outlined in this section and it is the “contract” partners need to abide to support integration moving forward. This will facilitate independent tool development, while accommodating collaboration across the consortium partners in meeting the XIVT project objective.

3.1 High-Level Service Attributes

As stated above, we first define the input and output attributes supported by the toolchain services (summarized in the diagram below). The diagram also cross refers to partner specific tools if one already exists.



		INPUT	OUTPUT	EXAMPLE TOOLS
A01 Variability Modelling		• Behavior Description • Parameter Range Information	• Variability Model Description	• 5.5 • 5.7 • 5.14
A02 Feature Reuse Analysis		• Variability Model Description	• Variability Model Description (with Similarity Clustering)	• 5.11 • 5.18
A03 Sampling		• Variability Model Description • Configuration Range	• Product Model Description (with generic classes)	• 5.7 • 5.16 • 5.20 5.15 5.19
A04 Instantiation		• Product Model Description (with generic classes)	• Product Model Description (with classes instantiated)	• (---)
A05 Test Modelling		• Product Model Description (with classes instantiated)	• Test Model Description	• (---)
A06 Test Reuse Analysis		• Test Model Description	• Test Model / Test Conditions (after Test Reuse Analysis)	• (---)
A07 Test Generation	A07.a Test Behavior Generation	• Test Model / Test Conditions	• Test Behavior Description *(at logical level)*	• (---)
	A07.b Test Structure Generation	• Test Model / Test Conditions	• Test Preconditions *(at logical level)*	• (---)
	A07.c Test Data Generation	• Test Model / Test Conditions	• Test Data *(at logical level)*	• (---)
A08 – Test Optimization	A08.a Test Priorization	• Logical Test Case List (with Preconditions and Test Data for each test case)	• Logical Test Case List (with priority score)	• 5.1 • 5.10 • 5.14
	A08.b Test Selection	• Logical Test Case List (with priority score)	• Logical Test Case List (test subset)	• (---)
A09 Test Adaptation		• Logical Test Cases *(also known as: Abstracted Test Cases)*	• Executable Test Cases *(also known as: Concrete Test Cases)*	• 5.2


A10
Test Execution
A10.a
Test Execution
List of Executable Test Cases
Test target system details
Optional parameters
(priority, schedule, etc)
Array of test results
(---)
A10.b
Test Behavior
Simulation
List of subsystems to simulate
Input data for subsystem to be simulated
Subsystem output
(---)
A10.c
Test Logging
Test case triggered for execution & other details
Execution history and outcome (pass/fail)
(---)

3.2 Detailed Service Attributes

This section outlined the standardized service definition in terms of inputs and outputs. It is important that partners think in terms of services / functionality rather than tools they are familiar with. This will ensure across the consortium there is alignment and clarity as to what each layer of the toolchain is tasked with. Therefore a tool from a partner is likely mapped to multiple services and inputs and outputs are defined in the context of the services from external trigger and response perspective.

1. Partners have identified which toolchain activities their tool(s) can support
2. Data was collected from partners as to what input and output their services currently support.
3. Collected data was analyzed to come up with a standard interface definition.
4. Next Steps:
    a. Using this standard definition as a guide, work with partners to identify gaps in terms interfaces that are missing, needs to be updated.
    b. Based on the gaps analysis, define a timeline for required changes to be implemented (partners have the option to do it themselves or can request help of other partners in the consortium that are specialized in software integration and product development)

c. At this point, we can start formal integration effort, leading to D4.4 (Integrated XIVT Toolchain)

3.2.1 A01 – Variability Test Case Modelling

Variability Test Case Modelling Service provides support for generating a set of test cases, mapped to the variants. This service requires the State Machine Model of the requirements as input and will generate a traceability matrix of test cases per protocol definition below. Requirements can have additional attributes such as textual description or images depending on the use case.

Standard Protocol

Activity / Service	Input Attribute	Input Type	Output Attribute	Output Type
A01	State Machine Model	JSON	Test Cases Matrix	JSON

3.2.2 A02 – Feature Reuse Analysis

Feature Reuse Analysis Service provides the ability to take a set of new product requirements and maps those requirements to features previously defined for existing products/products lines to create the new product configuration required to support the new set of requirements.

Standard Protocol

Activity / Service	Input Attribute	Input Type	Output Attribute	Output Type
A02	Training Data	JSON	trainedModel	JSON (as per 5.11, 1 Train)
	Query Data	JSON	Comparison report	JSON (as per 5.11, 2 Query)

3.2.3 A03 – Sampling

Sampling Service provides support for automatically finding valid configurations from a given variability model (e.g. feature model).

Standard Protocol

Activity / Service	Input Attribute	Input Type	Output Attribute	Output Type
A03	Variability Model	JSON	List of Configurations	JSON

3.2.4 A04 – Instantiation

Instantiation Service provides support for generating or selecting an instance (one particular object from a general description of a class of similar objects). Specifically, generating or selecting a concrete requirement or test case from an abstract form (i.e. the model) or incase of selection, from a pool of concrete requirements or test cases.

Standard Protocol

Activity / Service	Input Attribute	Input Type	Output Attribute	Output Type
A04	Variability Model	JSON	Set of concrete objects	JSON
	Set of concrete objects	JSON	Selected object	JSON

3.2.5 A05 – Test Modelling

Test Case Modelling Service provides support for generating a model set of test cases. This service requires the Requirements Model as input and will generate an output set of test cases per protocol definition below. Requirements can have additional attributes such as textual description depending on the use case.

Standard Protocol

Activity / Service	Input Attribute	Input Type	Output Attribute	Output Type
A05	List of textual requirements	JSON	List of Test Cases	JSON

3.2.6 A06 – Test Reuse Analysis

Test Reuse Analysis Service will provide a set of existing tests that can be used when testing similar products or new product features. In order to do so, the service requires a set of product/variant requirements and all the existing test cases for that specific type of product/product line/feature. Based on this, a list of reusable test cases would be identified from the pool of test cases that exist to adequately validate the new requirements.

Standard Protocol

Activity / Service	Input Attribute	Input Type	Output Attribute	Output Type
A06	List of Product requirements and List of Existing Test Cases	JSON	List of Reusable Test Cases	JSON

3.2.7 A07 – Test Generation

Test Generation Service includes 3 subprocesses: test behaviour generation, test structure generation and test data generation. Each of the specific sub-process is explained in its correspondent subsection and can be handled in parallel.

A07-A - Test Behavior Generation

Test Behaviour Generation Service will provide a set of configurable environment parameters required to emulate the expected behaviour of the testing system for a particular product / product line / feature of interest. This service requires the product / product line / feature of interest as an input, along with a list of dependent test system environment element(s). Based on this, the service will generate a list of configurable environment parameters applicable for the testbed(s) / test system(s).

Standard Protocol

Activity / Service	Input Attribute	Input Type	Output Attribute	Output Type
A07-A	Product / product line / feature of interest Test system dependency mapping	JSON	Environment configuration parameters	JSON

A07-B - Test Structure Generation

Test Structure Generation Service generates the pre-condition data required to setup and manage various test systems that are applicable for a particular product / product line / feature testing.

Standard Protocol

Activity / Service	Input Attribute	Input Type	Output Attribute	Output Type
A07-B	Product / product line / feature of interest Test system dependency mapping	JSON	Array of Test Structure Objects	JSON

A07-C Test Data Generation

Test Data Generation Service will provide a set of test data that will be used to test products or features. Test data means the possible input combination given a specific product/product line/feature.

Standard Protocol

Activity / Service	Input Attribute	Input Type	Output Attribute	Output Type
A07-C	product / product line / feature of interest mapping of product / product / feature list to related test cases	JSON	List of Test Data Objects	JSON

3.2.8 A08 – Test Optimization

Test Optimization has two subprocesses: test prioritization and test selection. Each of the specific sub-process is explained in its correspondent subsection and can be handled in parallel.

A08-A - Test Prioritization

Test Prioritization Service takes an input set of test cases and assigns priorities per internal knowledge-based algorithm and provides a prioritized set of test cases as output.

Standard Protocol

Activity / Service	Input Attribute	Input Type	Output Attribute	Output Type
A08-A	List of test cases	JSON	List of test cases with priority assignment	JSON

A08-B - Test Selection

Test Selection Service takes an input set of prioritized test cases and recommends a subset of test cases that are adequate to validate a feature / product / product line requirements per internal knowledge-based algorithm.

Standard Protocol

Activity / Service	Input Attribute	Input Type	Output Attribute	Output Type
A08-B	Prioritized List of test cases Feature / product / product line requirements Selection criteria (optional)	JSON	List of recommended test cases (sub-set)	JSON

3.2.9 A09 – Test Adaptation

Test Adaptation Service will adapt logical / abstract test cases to system specific/concrete test cases. All technical test details should be adapted by this service.

Standard Protocol

Activity / Service	Input Attribute	Input Type	Output Attribute	Output Type
A09	List of abstract test cases system under test parameters	JSON	List of Concrete Test Cases	JSON

3.2.10 A10 – Test Execution

Test Execution has three subprocesses: test execution, test behaviour simulation and test logging. Each of the specific sub-process is explained in its correspondent subsection and can be handled in parallel.

A10-A - Test Execution
Test Prioritization Service is responsible for executing a set of executable test cases per input provided and monitoring the status of execution to assert test pass/fail and return the results as output.

Standard Protocol

Activity / Service	Input Attribute	Input Type	Output Attribute	Output Type
A10-A	List of executable test cases Test target system details Optional parameters (priority, schedule, etc)	JSON	Array of test results	JSON

A10-B - Test Behavior Simulation

Test Behavior Simulation Service is responsible for supporting simulation of testbed (both software and hardware) to enable Test Execution when subsystems are not available (physical unavailability, need for stubs in software).

Standard Protocol

Activity / Service	Input Attribute	Input Type	Output Attribute	Output Type
A10-B	List of subsystems to simulate Input data for subsystem to be simulated	JSON	Subsystem output	JSON

A10-C - Test Logging

Test Logging Service is responsible for tracking test execution details and writing to a repository the execution history, including the execution outcome (pass/fail) to be persistent.

Standard Protocol

Activity / Service	Input Attribute	Input Type	Output Attribute	Output Type
A10-C	Test case triggered for execution & other details	JSON	Execution history and outcome (pass/fail)	JSON

3.2.11 A20 – Mutation Testing

Mutation Testing Service provides the ability to automate fault detection. In order to do so, the service takes the list of test cases to undergo detection and runs it on an original item and it's variants. The results produced are detected for faultiness and the details of the findings are returned as an output.

Standard Protocol

Activity / Service	Input Attribute	Input Type	Output Attribute	Output Type
A20	List of Test Cases List of Test Items and Variants	JSON	List of faults and Details of faults detected	JSON

3.2.12 A21 – Fault Injection

Fault Injection Service will inject algorithmically generated faults into the system once triggered.

Standard Protocol

Activity / Service	Input Attribute	Input Type	Output Attribute	Output Type
A21	Fault Injection enabled/disabled Fault injection duration	JSON	System generated faults	JSON

3.2.13 A30 a, b & c – Coverage Analysis (Requirements / Code / Feature Coverage)

Coverage Analysis Service will allow the user to identify the depth of coverage achieved by executing a particular test suite. The service runs verification on a predefined set of coverage items such as requirements, features or portions of code and returns coverage metrics as output.

Standard Protocol

Activity / Service	Input Attribute	Input Type	Output Attribute	Output Type
A30	List of Test Cases Type of Coverage to be analyzed (requirements, code, feature) Requirements / code / feature details	JSON	Test coverage metrics	JSON

3.2.14 A31 – Test Specification (informal, natural language)Validation

Test Specification Validation Service will take a set of test specifications specified as natural language in text format as input and validate it for completion against a predefined set of semantics. After processing, a validation status per test case will be provided as output.

Standard Protocol

Activity / Service	Input Attribute	Input Type	Output Attribute	Output Type
A31	Set of test cases Standard Test Case Semantics	JSON	List of test cases with validity status (valid / invalid)	JSON

3.2.15 A32 – Test Model Validation

Test Model Validation Service will take a Test Model as input and validate it for completion against a predefined set of attributes related to variability. After processing, a validation status model element will be provided with validity status.

Standard Protocol

Activity / Service	Input Attribute	Input Type	Output Attribute	Output Type
A32	Test model Mandatory attributes	JSON	Model attribute status (valid / invalid)	JSON

Appendix A: Tools and Toolchain Mapping

3.3 Introduction

This section reshapes the Mapping of Tools to Services designed in D4.2, section 4.3. This updated version maps to specific toolchain activities and not generic features/services. Using this new approach, we have also identified some incongruencies between concepts that this document tries to solve.

For better consolidation and information check, we have put the list of tools in the next matrix using the same order they are presented in Chapter 5 (being also the same order they were presented at D4.2, chapter 5).

This tool list is not finished as D4.2 also stated. However, at this D4.3 it became more evident that XIVT consortium needs to consolidate the information, in special because there are also some columns (abstract services) where there is no tool service identified.

Those columns could be addressed by new tools or by existing tools even though there is no service identified yet. For example, if we use the conceptual explanation in

Chapter 1.1, page , where a tool functionality was divided into three services, we can guess that this could happen from existing tools to address those empty columns.

3.4 Mapping of Tools to Toolchain Activities

The next page shows the above-mentioned matrix.

								Test Generation			Test Optimization						
	Tool	A01 Variable Modeling	A02 Feature Reuse Analysis	A03 Sampling	A04 Instantiation	A05 Test Modeling	A06 Test Reuse Analysis	A07.a Test Behavior Generation	A07.b Test Structure Generation	A07.c Test Data Generation	A08.a Test Priorization	A08.b Test Selection	A09 Test Adaptation	A10.a Test Execution	A10.b Test Behavior Simulation	A10.c Test Logging	A2 Muta Test
5.1	VarSel		+														
5.2	Service for Test Case Adaptation									x							
5.3	Service for Test Suite Quality Assessment																o
5.4	Service for Test Injection and Execution (FBDMutator)												+	+		+	x
5.5	SEAFOX	x				x		+	+	x	o	o					
5.6	RCM (NALABS)	o															
5.7	BeVR	+		+		+											
5.8	Service for generating human-readable test scripts							o									
5.9	Service for IIoT component identification					o	o										
5.10	Service for risk-based test scoring										+						
5.11	VARA		+														
5.12	IntegrationDistiller																
5.13	SaFReL														x		
5.14	ReForm	o															
5.15	ifakVBT			x		x											
5.16	MTest					+		+	+					+	+	+	
5.17	RELOAD														x		
5.18	TARA						o										
5.19	DoTA		o								o						
5.20	MODICA	x		x		x		x									
5.21	ARD Test Prioritization Tool					+		+			+	+				+	
5.22	MERAN	x			x												
5.23	TESTONA	x		x		x											
5.24	InnSpect											x	x	x		x	
5.25	DeltaFuzzer																+
5.26	TV RoboTester													x	x		
5.27	Otomat (Testroyer)					o					+	+		x		x	

x	Ready
+	In Development
o	Planned

Figure 4.1 - Mapping of Tools to Toolchain Activities

Appendix B: Detailed Tools and Attributes Definition

****
Var. Modelling
Sampling
Instantiation
Test Modelling
Test Data Generation (incl. Data Fuzzing)
Test Behavior Generation
Test Structure Generation
Test Prioritization
Test Selection
Test Execution
Test Adaptation
Test Logging
Mutation Testing (incl. Fault Injection)
Coverage Analysis (e.g. Req. / Code / Feature Coverage)
Test Behavior Simulation
Test Specification (informal, natural language)Validation
Test Model Validation
Test Reuse
Feature Reuse
****

This section explains, for each tool, what are the services (toolchain activities) that could be supported by it, and as well, the specificities of that tool that need to be addressed to be inserted into the toolchain and follow the standard protocol.

It is also important that a brief description of the associated tool be provided, so that all XIVT partners can understand the purpose of the tool. We know that some tools are industry oriented and could be applied only to specific use cases.

The suggested approach is to use the detailed attributes (see Chapter 3.2, page) and identify if there are conflicts with that specific tool, so that we can:

- Update the Detailed Attributes because we realize that it is missing some information
- Create an adaptor so that the specific tool can interact with the Toolchain Orchestrator (using the standard protocol)

Because this chapter depends from Chapter 3.2, page , it will be filled in after 2020, April 30th because it is impossible to finish Chapter 3.2 before than date.

3.5 VarSel: Service for Variant Selection

3.6 Service for Test Case Adaptation

This is not a separate service, but an application of BeVR (see 5.7 below).

Activity / Service	Input Attribute	Input Type	Output Attribute	Output Type
?	Test Base Model	XMI/UML/UTP	Test Model	XMI/UML/UTP
	Test Variability Model			
	Test Resolution Model			

The Test Base Model consists of a set of files. The basic file format is XML Metadata Interchange. In order to understand the semantics, however, knowledge of the Unified Modeling Language and UML Testing Profile is advisable.

The Test Variability Model ...

The Test Resolution Model ...

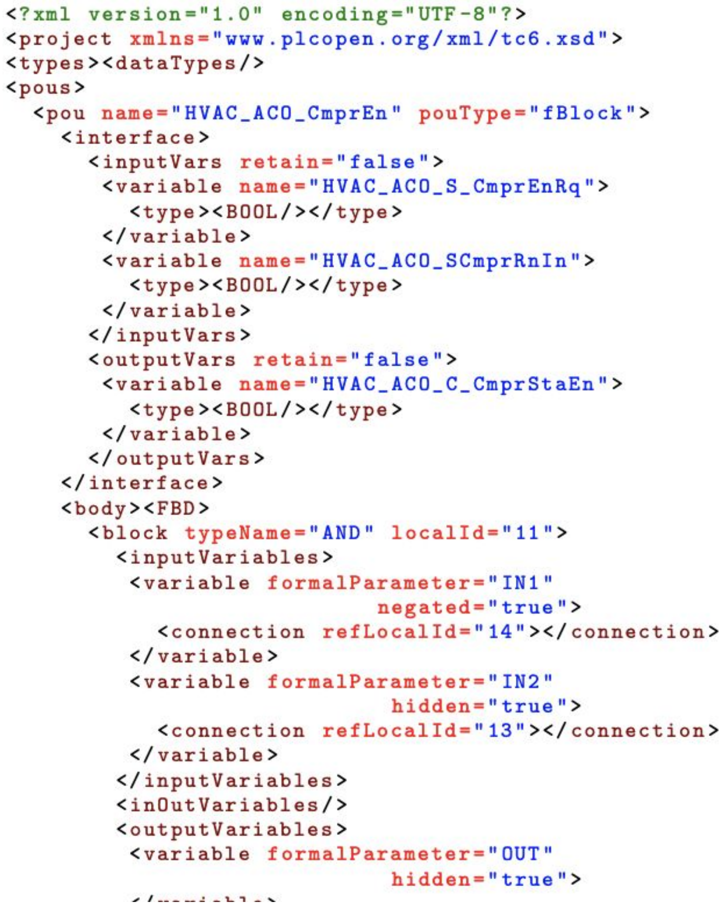
3.7 Service for Test Suite Quality Assessment

3.8 Service for Test Injection and Execution (FBDMutator)

Input

Variable	Type	Description
XML file	PLCOpen XML file (detail below)	The PLCOpen language is implemented as an XML profile that provides the ability to describe FBD programs using this profile. The PLCOpen language provides both structural and graphical information needed for implementing the actual translation.

An example of a PLC OpenXML file:



```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="www.plcopen.org/xml/tc6.xsd">
<types><dataTypes/>
<pous>
  <pou name="HVAC_ACO_CmprEn" pouType="fBlock">
    <interface>
      <inputVars retain="false">
       <variable name="HVAC_ACO_S_CmprEnRq">
         <type><BOOL/></type>
       </variable>
       <variable name="HVAC_ACO_SCmprRnIn">
         <type><BOOL/></type>
       </variable>
      </inputVars>
      <outputVars retain="false">
       <variable name="HVAC_ACO_C_CmprStaEn">
         <type><BOOL/></type>
       </variable>
      </outputVars>
    </interface>
    <body><FBD>
      <block typeName="AND" localId="11">
        <inputVariables>
         <variable formalParameter="IN1"
                           negated="true">
           <connection refLocalId="14"></connection>
         </variable>
         <variable formalParameter="IN2"
                            hidden="true">
           <connection refLocalId="13"></connection>
         </variable>
        </inputVariables>
        <inOutVariables/>
        <outputVariables>
         <variable formalParameter="OUT"
                            hidden="true">
         </variable>
```

Output

Variable	Type	Description
A set of XML files	Folder containing a set of PLCOpen XML files.	The tool generates XML files corresponding to the mutants created.

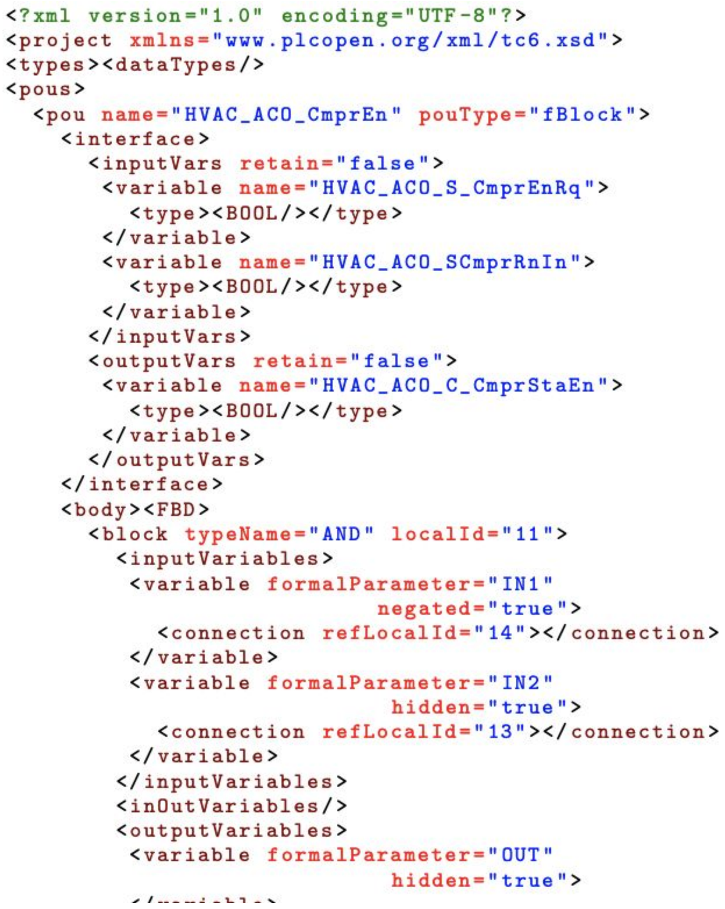
3.9 Service SEAFOX

Input

Variable	Type	Description
XML file	PLCOpen XML file (detail below)	The PLCOpen language is

		implemented as an XML profile that provides the ability to describe FBD programs using this profile. The PLCOpen language provides both structural and graphical information needed for implementing the actual translation.

An example of a PLC OpenXML file:



```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="www.plcopen.org/xml/tc6.xsd">
<types><dataTypes/>
<pous>
  <pou name="HVAC_ACO_CmprEn" pouType="fBlock">
    <interface>
      <inputVars retain="false">
       <variable name="HVAC_ACO_S_CmprEnRq">
         <type><BOOL/></type>
       </variable>
       <variable name="HVAC_ACO_SCmprRnIn">
         <type><BOOL/></type>
       </variable>
      </inputVars>
      <outputVars retain="false">
       <variable name="HVAC_ACO_C_CmprStaEn">
         <type><BOOL/></type>
       </variable>
      </outputVars>
    </interface>
    <body><FBD>
      <block typeName="AND" localId="11">
        <inputVariables>
         <variable formalParameter="IN1"
                           negated="true">
           <connection refLocalId="14"></connection>
         </variable>
         <variable formalParameter="IN2"
                            hidden="true">
           <connection refLocalId="13"></connection>
         </variable>
        </inputVariables>
        <inOutVariables/>
        <outputVariables>
         <variable formalParameter="OUT"
                            hidden="true">
         </variable>
```

Output

Variable	Type	Description
A CSV file	Test cases represented in a CSV file	The tool generates a CSV file corresponding to the test cases

		created.

3.10 Service RCM (NALABS)

Input

Variable	Type	Description
Requirements written in an XLXS file	XLXS file	A requirement document containing an ID and a text column.

Input

Variable	Type	Description
Results in an XLXS file	XLXS file	A requirement document containing the measurements for each Requirement ID.

3.11 BeVR: Service for requirements-based variability modelling

3.12 Service for generating human-readable test scripts

3.13 Service for IIoT component identification

3.14 Service for risk-based test scoring

3.15 VARA: Service for requirements similarity-based reuse prediction

End Points:

1. Train

Trains a model on natural language requirements and their reuse links to product line features. Note that the tool is console based and takes these input through terminal.

Input:

Variable	Type	Description
train	JSON(detailed next)	List of customer requirements and their links to product line features

‘train’ array JSON:

```
{
 "cust": [
  {
   "ID": "REQ123",
   "text": "string",
   "link": "PLF123"
  }
 ],
 "features": [
  {
   "ID": "PLF123",
   "text": "string"
  }
 ]}
```

Output:

Variable	Type	Description
model	JSON(Detailed next)	Trained model

Trained model ‘model’ JSON:

```
{
  "status": "string",
  "model": [
   {
     "modelName": "string",
     "vectorSize": 300,
     "accuracy": 0.0,
     "path": "string"
   }
  ]
```

2. Query

Queries a trained model on natural language requirements and produces similarity and reuse report.

Input:

Variable	Type	Description
query	JSON(detailed next)	Model to be queried, list of customer requirements

The 'query' JSON:

```
{
  "modelName": "string",
  "query": [
   {
     "ID": "REQ123",
     "text": "string"
   }
  ]
}
```

Output:

Variable	Type	Description
comparisions	JSON(detailed next)	Similarity and reuse report

The 'comparison' JSON represent the similarity and reuse report:

```
{
  "comparisons": [
   {
     "ID": "string",
     "max_sim": 0.00,
     "top_reuse": "PLF123",
     "reuse":[{"based_on":"string", "sim":0.0, "reuse_candidate":"string"}]
   }
  ]}
```

3.16 SaFReL: Service for performance test case generation

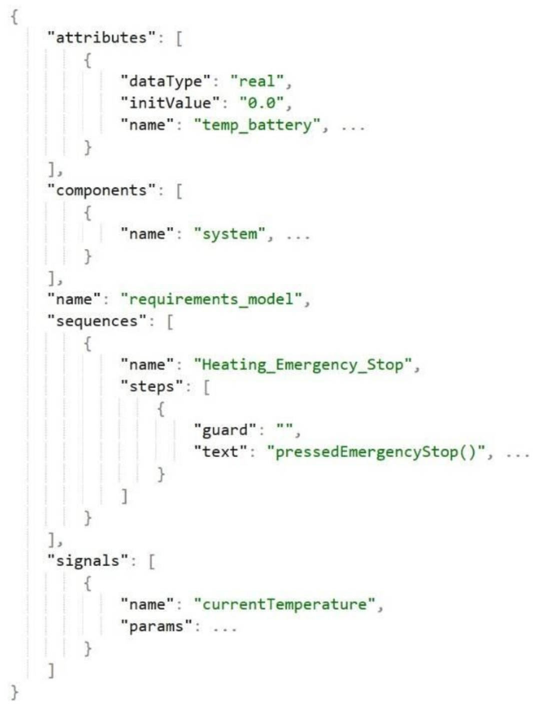
3.17 ReForm: Service for requirement formalization and test optimization

Input

Variable	Type	Description
requirements.json	JSON (not yet defined/under development)	List of textual requirements

Output

Variable	Type	Description
requirements_model.json	JSON (see example below)	Requirement models in IRDL (ifak Requirements Description Language)
requirements_metrics.json	JSON (not yet defined/under development)	List of risk and use case metrics (table of values/percentages)



```
{
    "attributes": [
        {
            "dataType": "real",
            "initValue": "0.0",
            "name": "temp_battery", ...
        }
    ],
    "components": [
        {
            "name": "system", ...
        }
    ],
    "name": "requirements_model",
    "sequences": [
        {
            "name": "Heating_Emergency_Stop",
            "steps": [
                {
                    "guard": "",
                    "text": "pressedEmergencyStop()", ...
                }
            ]
        }
    ],
    "signals": [
        {
            "name": "currentTemperature",
            "params": ...
        }
    ]
}
```

3.18 ifakVBT: Service for test case generation and variant traceability

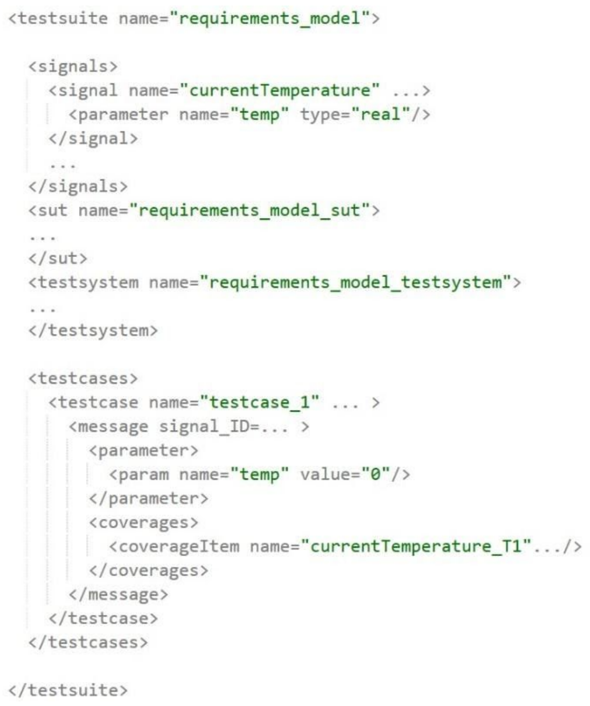
Input

Variable	Type	Description
requirements_model.json	JSON(see example in 5.14)	Requirement models in IRDL (ifak Requirements Description Language)

Output

Variable	Type	Description
statemachine.json	JSON (similar to example in 5.14)	UML state machine (ifak internal language)
testcases.xml	XML (see example below)	List of generated abstract test cases (ifak internal language)

trace.json	JSON (not yet defined/under development)	Traceability matrix of test cases to variants



```
<testsuite name="requirements_model">

  <signals>
    <signal name="currentTemperature" ...>
      <parameter name="temp" type="real"/>
    </signal>
    ...
  </signals>
  <sut name="requirements_model_sut">
  ...
  </sut>
  <testsystem name="requirements_model_testsystem">
  ...
  </testsystem>

  <testcases>
    <testcase name="testcase_1" ... >
      <message signal_ID=... >
        <parameter>
          <param name="temp" value="0"/>
        </parameter>
        <coverages>
          <coverageItem name="currentTemperature_T1".../>
        </coverages>
      </message>
    </testcase>
  </testcases>

</testsuite>
```

3.19 MTest

3.20 RELOAD: Service for test load generation

3.21 TARA

End points:

1. **Test Reuse Analysis**

Performs a parsing, classification and recommends existing test cases for reuse.
Input: Single entry from comparsions JSON ARRAY from VARA and Simulink Models

Note that this is a planned tool in XIVT tool-chain and the final end-ponits may differ.

Variable	Type	Description
comparision	JSON(detailed next)	Similarity and reuse report
testHarness	Simulink Model	Simulink model file of the target test cases for reuse analysis
implementationModel	Simulink Model	Simulink model file of the implementation of source requirement

The ‘comparison’ JSON represent the similarity and reuse report (in this case only one entry from the array is used):

```
{
    "ID": "string",
    "max_sim": 0.00,
    "top_reuse": "PLF123",
    "reuse":[{"based_on":"string", "sim":0.0, "reuse_candidate":"string"}]
}
```

Output:

3.22 DoTA

1. **Delta based test prioritization**

Performs test prioritization and selection based on coverage to delta.

Variable	Type	Description

Standard Model	Simulink Model	Simulink model of standard from product line
Variant Model	Simulink Model	Simulink model of with modifications

Output:

Variable	Type	Description
Delta tests	JSON(detailed next)	Delta coverage based selected test cases

The ‘deltaCases’ JSON represent the test reuse report:

```
{
 "selectedTests": [
  {
    "testName": "string",
    "predicted_coverage": 0.00
  }
 ]}
```

3.23 MODICA

3.24 ARD Test Prioritization Tool

3.25 MERAN

3.26 TESTONA

3.27 InnSpect

End-Points:

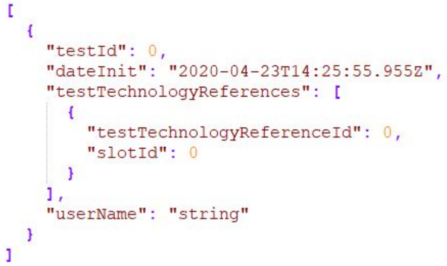
1- Schedule Tests

Schedules a list of tests HTTP Post.

Input

Variable	Type	Description
sch	JSON(detailed next)	List of tests to be scheduled with initial datetime of each test

sch array JSON description:

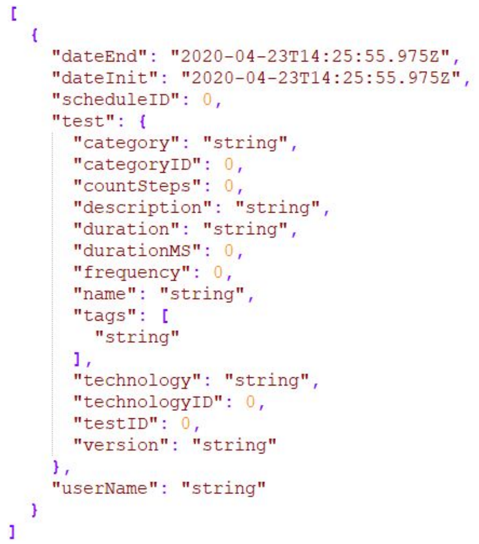


```
[
  {
    "testId": 0,
    "dateInit": "2020-04-23T14:25:55.955Z",
    "testTechnologyReferences": [
      {
        "testTechnologyReferenceId": 0,
        "slotId": 0
      }
    ],
    "userName": "string"
  }
]
```

Output

Variable	Type	Description
Scheduled tests	JSON(Detailed next)	List of scheduled tests

Scheduled tests array JSON description:



```
[
  {
    "dateEnd": "2020-04-23T14:25:55.975Z",
    "dateInit": "2020-04-23T14:25:55.975Z",
    "scheduleID": 0,
    "test": {
      "category": "string",
      "categoryID": 0,
      "countSteps": 0,
      "description": "string",
      "duration": "string",
      "durationMS": 0,
      "frequency": 0,
      "name": "string",
      "tags": [
        "string"
      ],
      "technology": "string",
      "technologyID": 0,
      "testID": 0,
      "version": "string"
    },
    "userName": "string"
  }
]
```

3.28 DeltaFuzzer

Input

Variable	Type	Description
config_file	A makefile containing the different targets to compile the target application.	Name of the configuration file to compile the target application. It is a makefile which can contain the target for executing the DeltaFuzzer
deltafuzzer_cmd	Text command line	Command line for executing the DeltaFuzzer. If the makefile contains a target that does this, this input variable is not needed. This command lines contain as arguments the in_directory and out_directory.
in_directory	String	Name of the input directory containing one or more test cases (inputs) to start testing the target application with the tool. Depending on the goal of the target application, these files can be from different formats (e.g., jpg, pdf. txt).

Output

Variable	Type	Description
out_directory	String	During the DeltaFuzzer execution, it generates dynamically other testcases. The output is the test cases that hang and crash the target application, and coverage more execution paths of the application allow. These tests

		are stored in directories under the out_directory specified.

3.29 TV RoboTester

TV RoboTester is the name of our test automation software. We are automating TV user tests with this tool. The program's main functionality is to create testcases manually and execute these testcases. There is a test oracle that decides if a testcase fails or passes.

During our XIVT effort, we will develop a functionality to use a list of TV OSD images to create a system model automatically. This may serve as a separate app that can be used by any partner or 3rd parties.

We will use this app/feature and adapt it to our TV RoboTester software so that our testcases (within TV RoboTester) will be re-written over this software model. So when the model changes, we won't need to change our testcases. That is our goal within XIVT project.

End-Points:

1- **Create System/Variant Model**

Creates system/variant model from a list of system images.

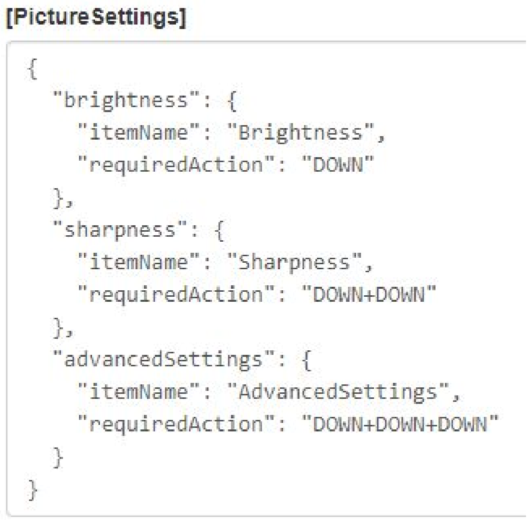
Input

Variable	Type	Description
OSD images of the TV software	PNG	List of OSD (on screen display) images of a TV software to create a system model

Output

Variable	Type	Description
System/Variant Model	JSON	System/Variant Model

System/Variant Model <sample> JSON description:



[PictureSettings]

```
{
  "brightness": {
    "itemName": "Brightness",
    "requiredAction": "DOWN"
  },
  "sharpness": {
    "itemName": "Sharpness",
    "requiredAction": "DOWN+DOWN"
  },
  "advancedSettings": {
    "itemName": "AdvancedSettings",
    "requiredAction": "DOWN+DOWN+DOWN"
  }
}
```

3.30 Otomat (Testroyer)

Appendix C: Technology Stack

As shown in the previous sections, each service can be independently developed. Nevertheless, for integration purposes we recommend the following initial technology stack to be used by services developed in WP2 and WP3:

- API Gateway - Kong
- Service Definition - Java or NodeJS
- Front End - AngularJS
- Storage - Cassandra or MongoDB

In the end this technology stack is a recommendation for helping developers to integrate their software services while maintaining compatibility with continuous integration practices and tools since the XIVT platform embraces a microservice like architecture, collectively providing facilities for the users to deploy testing services.