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	D4.2: High Level Platform Architecture - Rev 2		
Deliverable type	Report (R)		
Dissemination level	Public	Date	2020-03-31
Mapped tasks	T4.1: Define high level architecture for end to end test orchestration platform		

Summary

This report defines the various services and features, which are expected to be supported by the toolchain. Further, it maps the features to the tools built by various partners, that offer those services. Next, each tool is defined in detail elaborating on the functionality, architecture diagram, end points and dependencies, if any. Finally, an attempt is made to recommend a technology stack that would help the various independent tools to work in sync and serve as the backbone of the final toolchain.

It is important to note, that the list of tools provided in this report is not complete, and it is expected that more tools will be contributed in to the XIVT project as the project, and thereby those tools mature further. However, the available list, shall allow to define the toolchain baseline.

Next steps would be to build upon the baselining and identify challenges in collaborating the tools. Those challenges would then provide direction in defining the interaction guidelines, i.e. the supported end-points by the toolchain. Tools that will come to maturity at a later stage of the project shall need to respect those guidelines in order to be successfully integrated with the toolchain. These API definitions and guidelines around end-points will constitute the part of next deliverable, i.e. D4.3.

Table of Contents

1. The High Level Architecture Direction: Microservices

Traditional application design is often called “monolithic” because the whole thing is developed in one piece. Even if the logic of the application is modular it’s deployed as one group, like a Java application as a JAR file for example. This type of architecture is convenient because it all happens in one spot, however not practical for a project like XIVT that has many different partners working on different pieces of the reference tool chain implementation. As shown in Figure 1, Microservices separates all of the major parts of this monolith from each other, untangling the codebase and drastically changing how partners can collaboratively develop and contribute to the reference tool chain. With microservices, different functions (not literally functions, but functional parts of the tool chain) are all separate. They communicate with the user interface, each other, and instances of the database. Each independent service is packaged as an API so it can interact with the rest of the application elements.

The Microservices Architecture has a number of important benefits. First, it tackles the problem of complexity. It decomposes what would otherwise be a monstrous monolithic application into a set of services. While the total amount of functionality is unchanged, the application has been broken up into manageable chunks or services. Each service has a well-defined boundary in the form of an RPC- or message-driven API. The Microservices Architecture enforces a level of modularity that in practice is extremely difficult to achieve with a monolithic code base. Consequently, individual services are much faster to develop, and much easier to understand and maintain. Second, this architecture enables each service to be developed independently by each XIVT partner that is focused on that service. The XIVT partner is free to choose whatever technologies make sense, provided that the service abides by the API contract. Third, the Microservices Architecture enables each microservice to be deployed independently. Partners never need to coordinate the deployment of changes that are local to their service. These kinds of changes can be deployed as soon as they have been tested by each partner. Finally, the Microservices Architecture enables each service to be scaled independently, making it a robust enterprise grade approach.

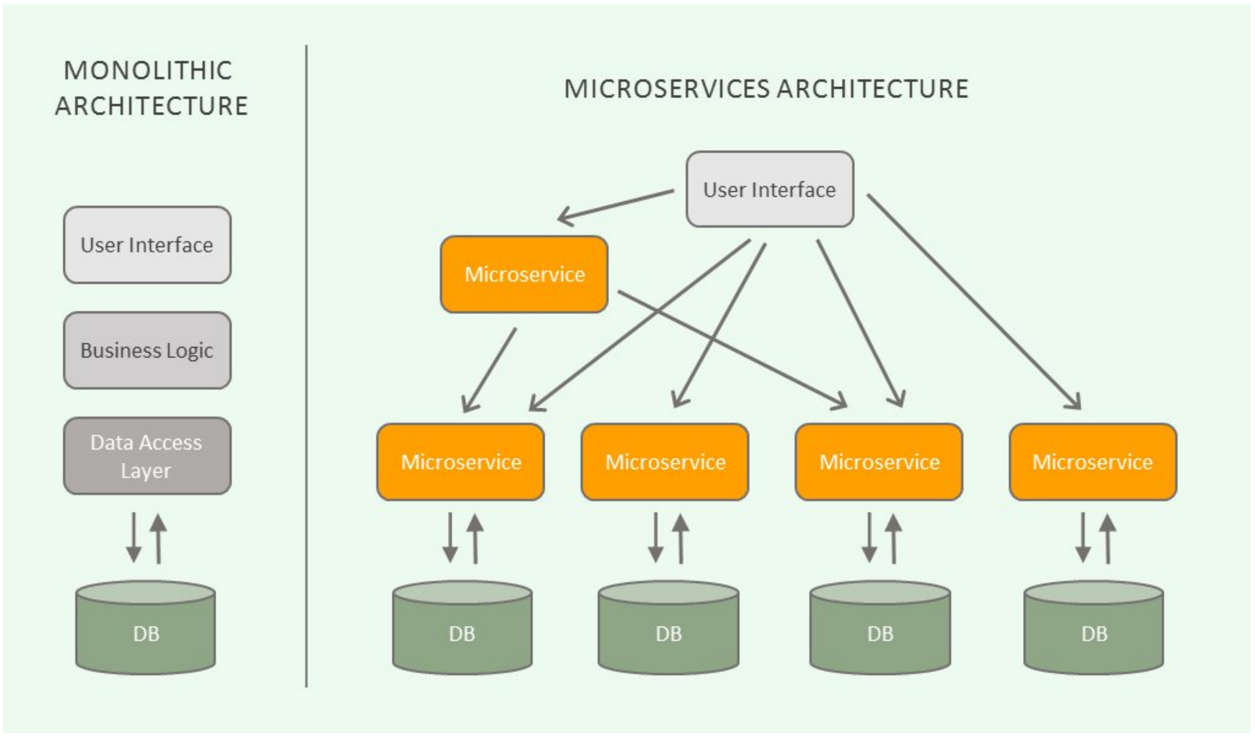
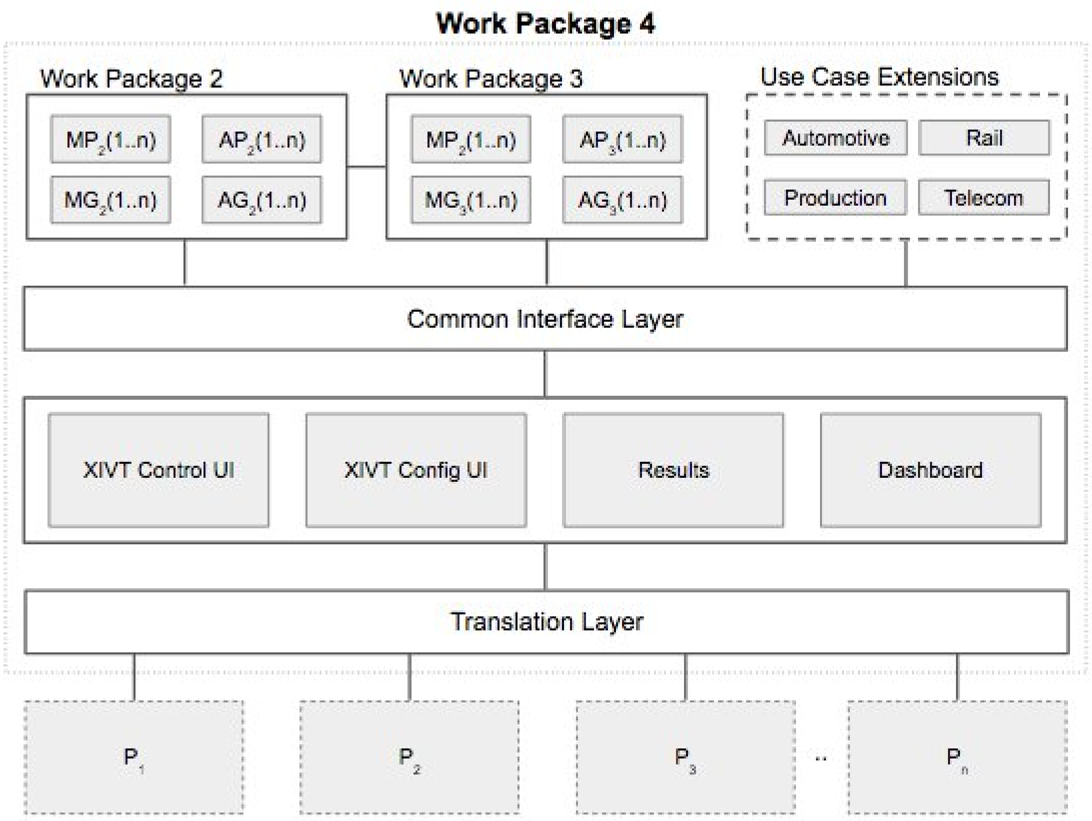


Figure 1. A brief comparison between the monolithic and microservices architectures.

The current report is the initial work for the XIVT architecture elaboration in a microservice architecture approach. In the next sections we establish the common conceptual infrastructure for all the architectural work and traceability and we identified initial technology requirements from different partners and we extracted high-level interfaces and deployment requirements.

2. Revisiting the XIVT Tool Chain Platform

The XIVT architecture has its basis in the release mechanism as integrated tool chains, which implements the proposed methods in a coherent way, and are available as a platform. The XIVT workbench shown in Figure 2 will allow to specify variant rich systems on the domain level, build instances, construct test cases and assess test suites in one place, across various use cases. The architecture of the tools would be such that the core platform can be easily extensible for any future use cases not explicitly addressed in the XIVT project.



MG - Module General
AG - Algorithm General
MP - Module Partner Specific
AP - Algorithm Partner Specific
P - Partner Product Extension

Figure 2. XIVT Initial Workbench.

External tools brought as background IPs by the consortium partners will be included by appropriate import and export-modules and need to support interface definitions of XIVT.

The output of this XIVT architecture will be an ecosystem of services around the proposed methodology. The following initial types of services based on the XIVT technologies were set to be implemented:

- Construction of test suites for highly configurable systems, an assessment of test suites.
- Certification support for safety-critical systems,
- Security analysis and testing, and
- Incremental enhancement of test suites for regression testing.

3. Breakdown of Functional Areas in the XIVT Platform

In this live document, we describe the XIVT Framework and its functional parts. The description is done by following a common pattern. We describe the high-level purpose of each UI and services and then we outline the functional interfaces that help to figure out the main features and possible means for integration. In addition, we briefly detail the subordinates – the constituent parts of each service in terms of the tools developed.

3.1 XIVT Control UI - Top Level Diagram

The XIVT framework regroups several interconnected tool sets including tool sets specifically developed in certain use cases. These tool sets are highly interconnected to achieve the goal of providing several services to the end user.

This is achieved using the XIVT control UI outlined in Figure 3 in which different microservices (e.g., Use Cases, Scheduler) are choreographed. Since there is no central coordination, each service produces and listens to events and decides if an action should be taken or not given the end user needs. In this way each service can coordinate their activities and processes to share information and value to the Control UI.

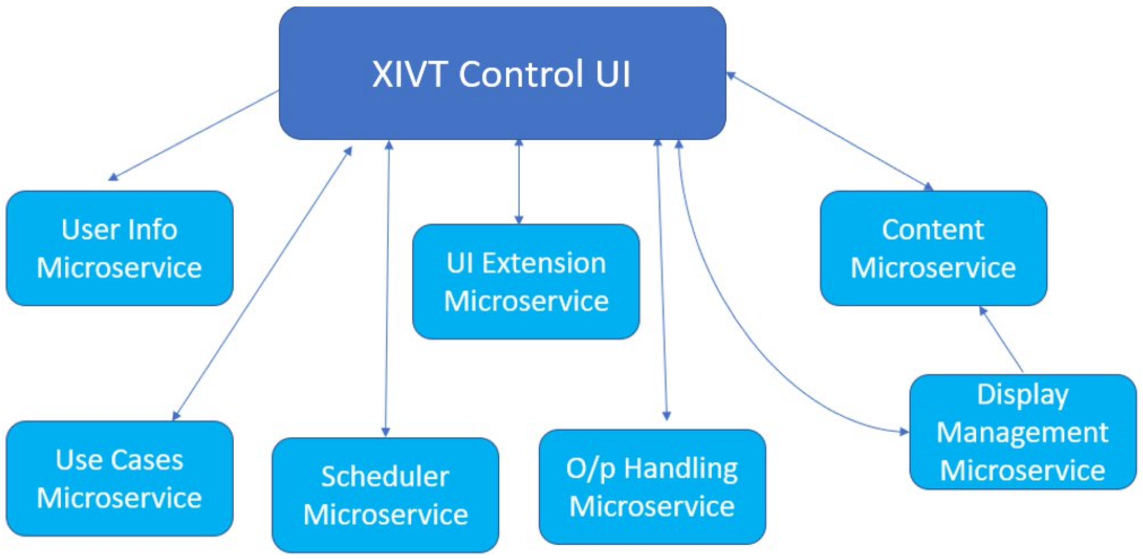


Figure 3. XIVT Control UI Structure.

Given this structural decomposition of the XIVT control UI, the management and communication between the end user and services can be viewed as a direct client-to-microservice communication architecture (shown in Figure 4) in which the end user can make requests directly to some of the microservices through services within the Docker host or internal cluster and to implement several cross-cutting concerns in an API Gateway (i.e., using the XIVT control UI structure).

In addition, this management and communication architecture is helping the use case and tool providers to get an understanding of how APIs are being used and how they are performing. This can be extended by providing real-time analytics reports and identifying trends in the usage of these microservices, including logs about request and response activity for further online and offline analysis.

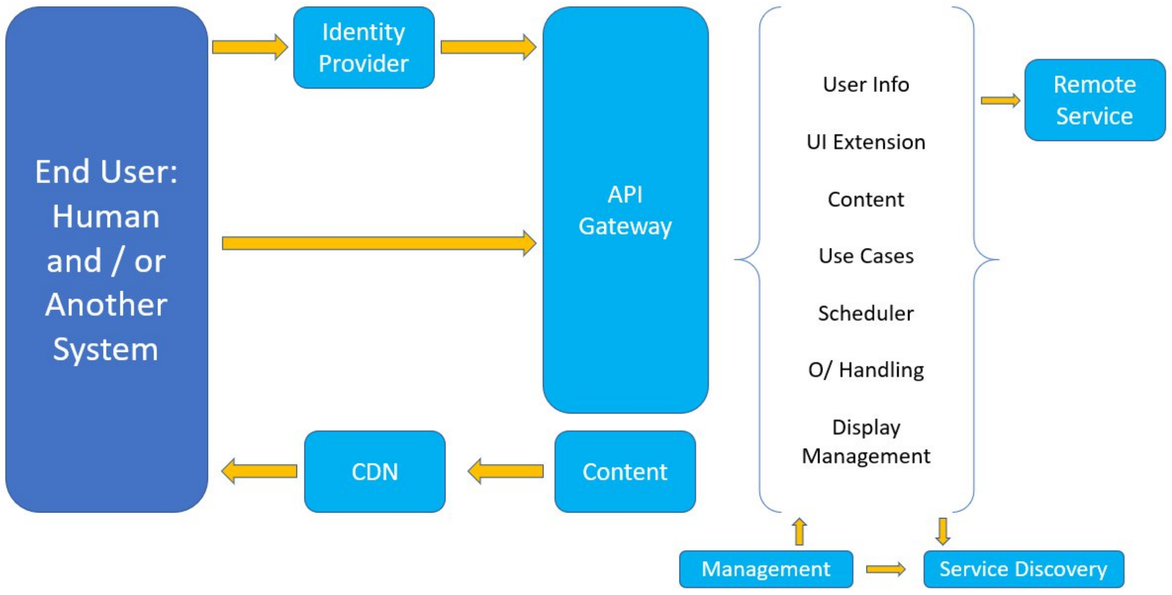


Figure 4. XIVT Framework API management and interaction between end users and service providers.

One of the primary challenges with microservices architecture is allowing services to discover and interact with each other as well as managing the service discovery. Since the XIVT framework will support the distributed characteristics of microservices architectures. Using meta information, such as configuration data, that can be used by each use case, the work in WP4 will explore several techniques for performing service discovery for microservices-based architectures.

In the next sections, we are outlining the WP2 and WP3 services that are developed for test optimization, generation and execution. In this iteration we have focused on developing, based on our initial tool designs, proof and concepts as well as performing adaptations to the baseline technologies in order to identify and start solving the integration issues in this XIVT framework. Hence, in an initial stage most services related to WP2 and WP3 only implement a small subset of the requirements.

The main use case of integrating different XIVT services is to allow users to manage jobs to execute different functionalities. For example, a tester can specify test jobs, execute them and analyze its results and the main information gathered execution are test results, logs and metrics. The XIVT framework UI should provide to the user a rich user interface specifically tailored to ease the management of that kind of information for each service provided by WP2 and WP3.

3.2 XIVT Test Optimization - Top Level Diagram

Besides all the features provided by XIVT UI than can be used by testers in the web interface, XIVT WP2 provides services that can be managed directly when a test suite is available. These services are shown in Figure 5 and allow a tester to select and prioritize tests with ease.

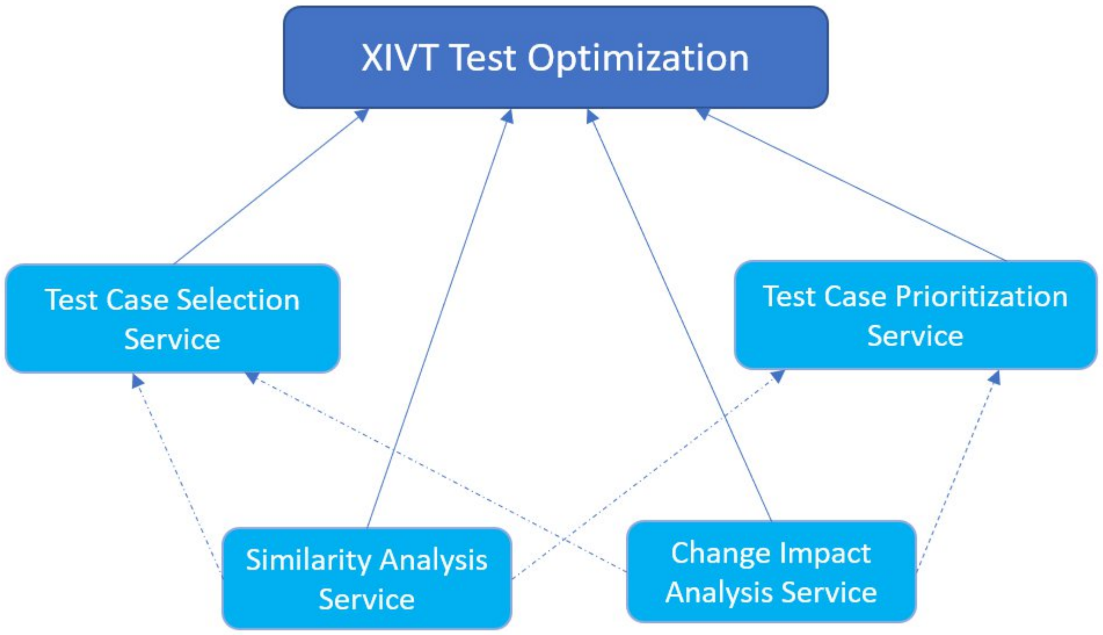

XIVT Test Optimization
Test Case Selection Service
Test Case Prioritization Service
Similarity Analysis Service
Change Impact Analysis Service

Figure 5. XIVT Test Optimization Services.

As XIVT test optimization services are the main entry point of XIVT framework features, it requires all other use case extensions and configurations to work properly. That is, this component requires all the use case specific components to be available. In the next section we outline the component that allows the XIVT framework to instrumentalize already deployed SUTs when services are executed against it.

3.3 USE Case Extension: Telecom - Top Level Diagram

The use case extension for the telecom domain (as shown in Figure 6) is implemented as an independent process communicated with other components using remote protocols or

APIs. Specifically, the Android Device Bench is used for providing a bench initiator, an execution manager and a logger among others.

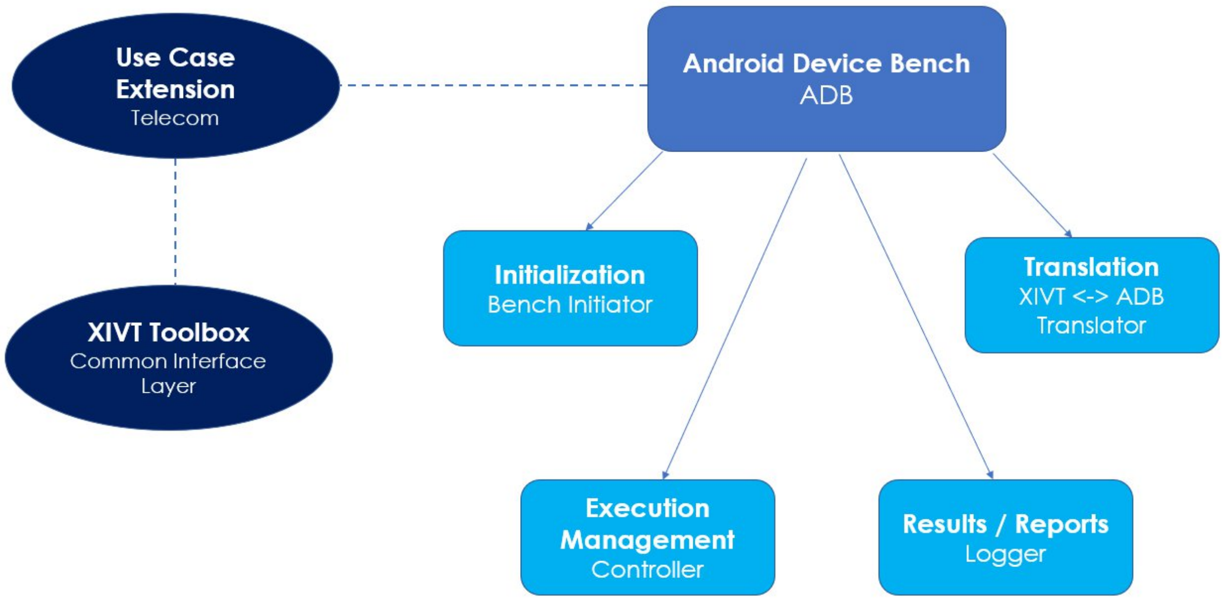


Figure 6. XIVT Use Case Extension for Telecom Domain.

This use case extension for the Telecom domain will be used in the XIVT Framework as a toolbox having a common interface layer that can be used by other use cases.

3.4 USE Case Extension: Railway - Top Level Diagram

In the case of the railway domain, the XIVT common interface toolbox will interact with the services provided by WP2 and WP3. For example (as shown in Figure 7), a service called VARA provided by WP2 will interact with the DOORS database and the models provided in MATLAB Simulink for requirement reuse analysis as well as generating and selecting test cases.

A remote API in this case is used by external tools (like a Jenkins plugin). In this case, a service can be executed multiple times and provide results in relation to the railway use case.

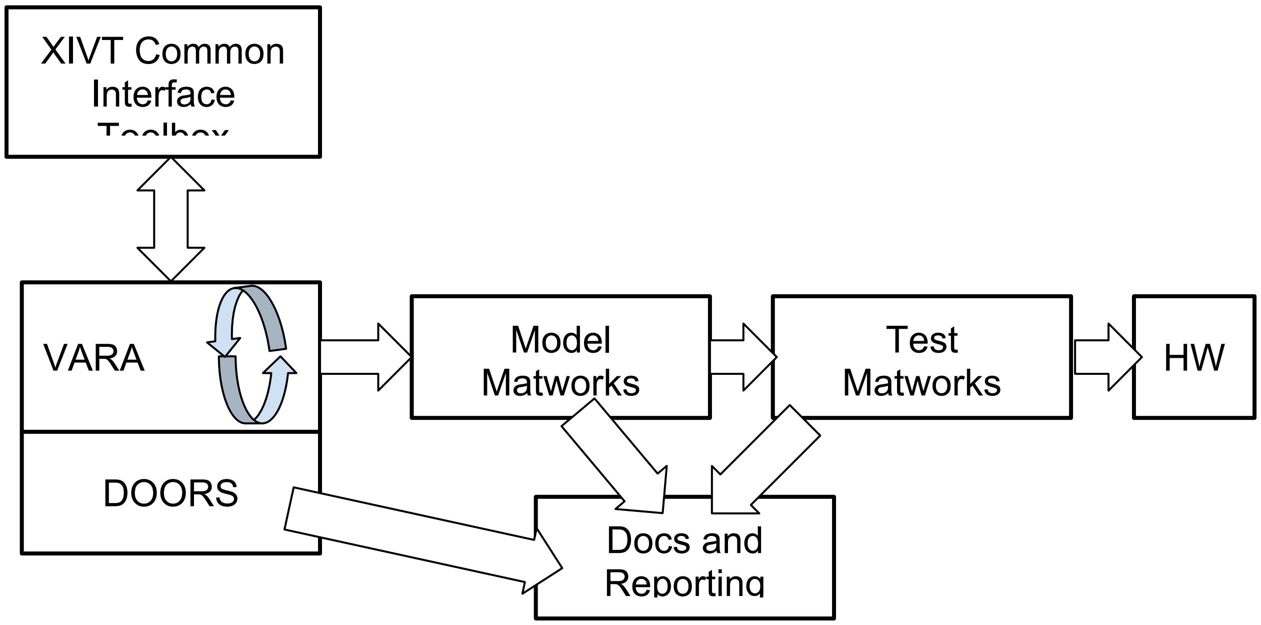
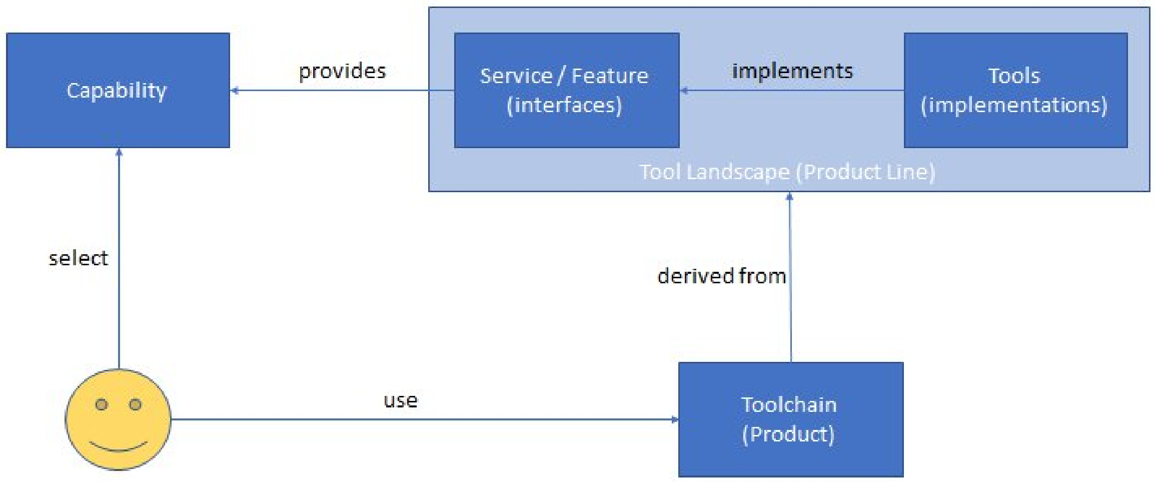


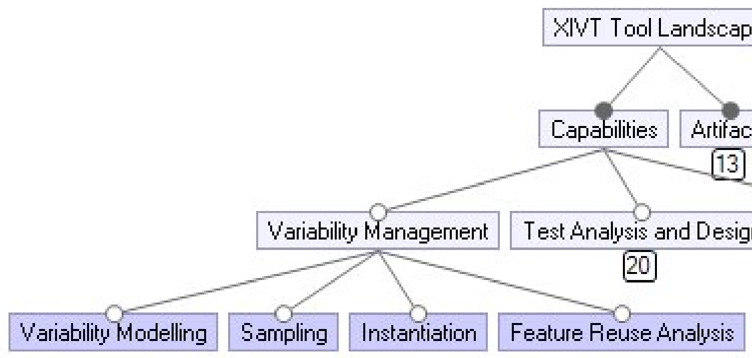
Figure 7. XIVT Use Case Extension for Railway Domain.

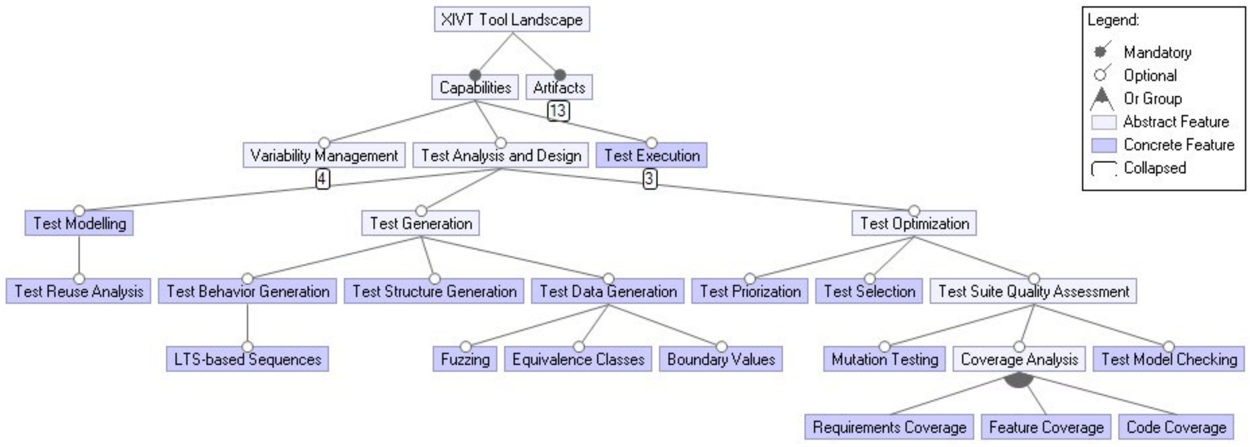
4. Features and Generic Services

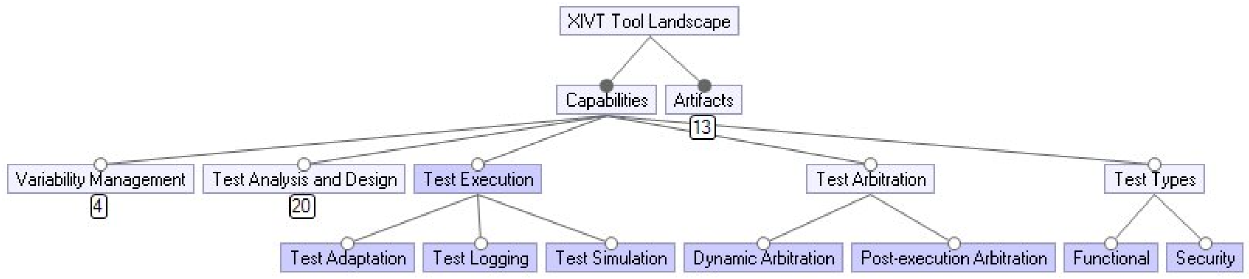
4.1 Introduction

In order to account for the variability in the different use cases with respect to processes, existing tools and data formats, the XIVT toolchain is designed as a product line itself. This product line is also referred to as the tool landscape. By selecting a set of capabilities from a capability model, a user can configure the tool landscape and thereby derive a concrete toolchain, tailored to her needs. The tool landscape consists of generic services or features, that provide certain capabilities. Each such service or feature can then be implemented by one or more concrete tools or services. The resulting toolchain consists of only those concrete tools or services which are applicable in any specific use case.




XIVT Tool Landscape
Capabilities
Artifacts
13
Variability Management
Test Analysis and Design
20
Test Execution
3
Variability Modelling
Sampling
Instantiation
Feature Reuse Analysis


XIVT Tool Landscape
Capabilities
Artifacts
13
Variability Management
4
Test Analysis and Design
Test Execution
3
Test Modelling
Test Generation
Test Optimization
Test Reuse Analysis
Test Behavior Generation
Test Structure Generation
Test Data Generation
Test Priorization
Test Selection
Test Suite Quality Assessment
LTS-based Sequences
Fuzzing
Equivalence Classes
Boundary Values
Mutation Testing
Coverage Analysis
Test Model Checking
Requirements Coverage
Feature Coverage
Code Coverage
Legend:
Mandatory
Optional
Or Group
Abstract Feature
Concrete Feature
Collapsed


XIVT Tool Landscape
Capabilities
Artifacts
13
Variability Management
4
Test Analysis and Design
20
Test Execution
Test Arbitration
Test Types
Test Adaptation
Test Logging
Test Simulation
Dynamic Arbitration
Post-execution Arbitration
Functional
Security

4.2 Feature Description

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

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

Instantiation (Synonyms: Product Derivation, Realization): Building or selecting an *instance*, that is, one particular object from a general description of a class of similar objects. In testing, building a concrete test case from an abstract one is a form of instantiation.

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.

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

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.

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

Test Data Generation: Sub-process of Test Generation related to test data.

Test Behavior Generation: Sub-process of Test Generation related to test behavior.

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

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.

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

Test Execution: Executing a test suite or test case, either in simulation or against the real test item.

Test Behavior Simulation: Execution a test suite against a simulated version of the test item.

Test Adaptation: Translating a logical test cases to executable, technical test cases.

Test Logging: Capturing relevant information during test execution, e.g., stimuli sent to and responses received from the test item.

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).

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

Coverage Analysis: Determining the degree in to which a test suite covers, i.e. verifies, a given set of coverage items, e.g., requirements, features or portions of code.

Requirements Coverage Analysis: Determining the degree in to which a test suite covers a set of requirements.

Feature Coverage Analysis: Determining the degree in to which a test suite covers a set of feature.

Code Coverage Analysis: Determining the degree in to which a test suite covers a code base.

Formal Verification: Proving or disproving the correctness of the test specification or test model with respect to a certain formal specification or property, using formal methods, e.g. through Model Checking.

4.3 Mapping of Tools to Features/Services

,	Services																		
NOTE: **x = Ready** **+ = In Development** **0 = Planned** **? = Not specified**	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
VarSel								+	+					+					
SEAFOX	x			x	x	+	+	0	0										
BeVR	+	+	+	+															
IntegrationDistiller																			
SaFReL					x					0					x				
ReForm	0			+				+											
ifakVBT	0			x		x		+											
NALABS (fka. RCM)	0															x	+		
TV RoboTester	0									+									
VARA																			+
Generation of human-readable test scripts										0		0							
IIoT component identification						0	0	0	0					0					
Risk-based test scoring			+					+	+										
Tool for test suite quality assessment													0	0					
MTest				+		+	+		+	+		+		+					
RELOAD					+					x		x							
FBDMutator										+	+	+	x						
TARA																		0	
DoTA								0	0					0					
MODICA	x	x		x		x		x	x					x			?		
MERAN	x		x																
TESTONA	x	x		x				x	x					x					
InnSpect										x	x	x							
DeltaFuzzer					+					+	+	x							
ARD Test Priorization Tool								0	0	0									
Test Case Instantiation																			
Otomat/Testroyer					0			+	+	x		x		x					

5. Detailed Tools and Services Definition

In this section we provide the initial definition of services provided by different partners by focusing on the category of service needed, the architecture diagram of the service, the endpoints as well as dependencies and the technology stack.

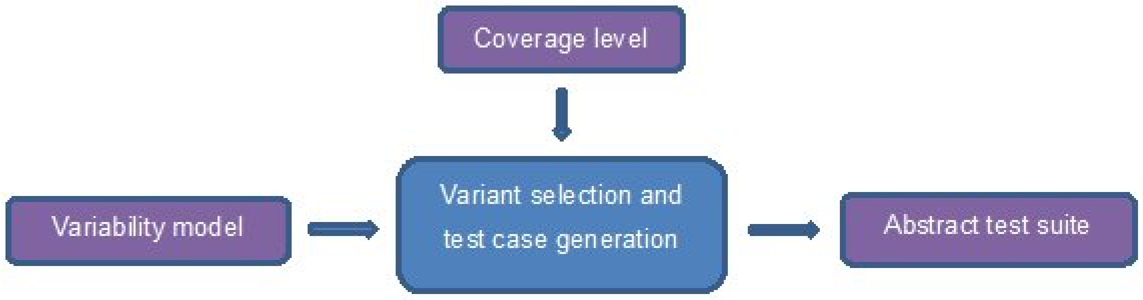
5.1 VarSel: Service for Variant Selection

Owner: Expleo

Category:
This tool is part of the tool for variant selection and test case instantiation, therefore of deliverable D3.3 of T3.2.

Description:
The functionality of the tool comprises the generation of test cases from a variability model of the SUT. The user first chooses a coverage level. The tool then generates test cases and chooses variants by means of the methods developed in XIVT, in particular in D3.2. There is also an algorithm planned to decide how close two variants are and whether and in which extent retesting is necessary.

Architecture diagram:



Endpoints:
The program takes the variability model from D3.2 as an input and allows further user input of (at least) coverage level. The output is a list of abstract test cases that ensures the requested coverage level.

Dependencies:
Tool for modelling variability, tool for test case instantiation, possibly UI of platform

Technology Stack: Java, Eclipse RCP, Angular JS

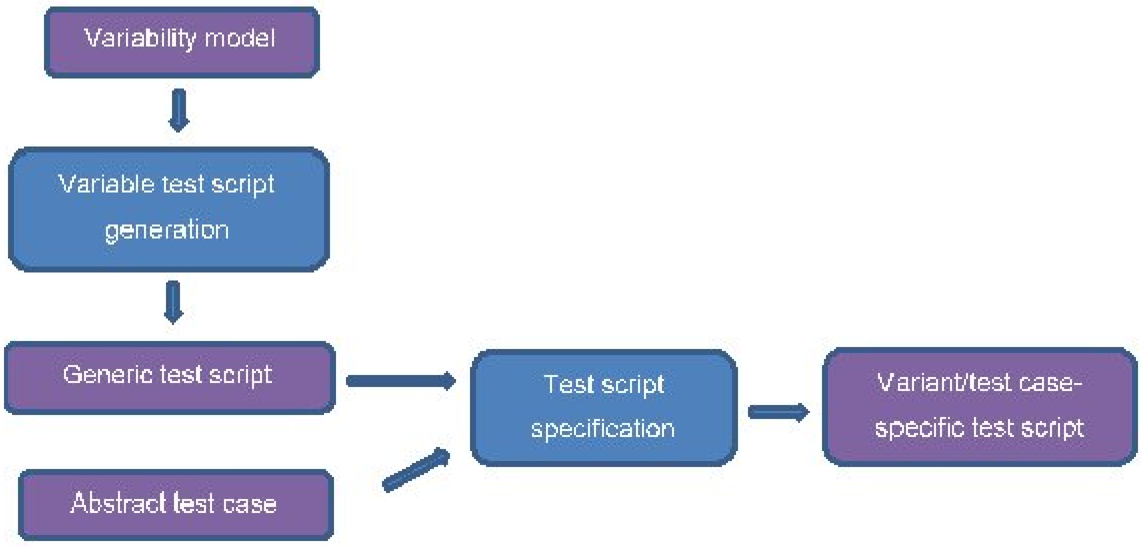
5.2 Service for Test Case Instantiation

Owner: Expleo

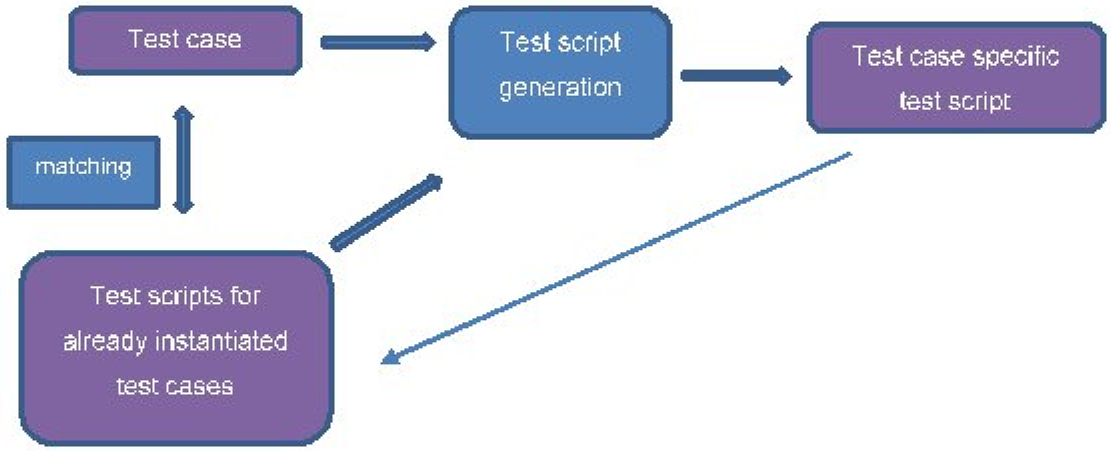
Category:
This tool is part of the tool for variant selection and test case instantiation, therefore of deliverable D3.3 of T3.2.

Description:
The functionality of the tool comprises the instantiation of the abstract test cases from the tool for variant selection above by means of the methods developed in XIVT, in particular in D3.2. Possible architectures could be:
- Pre-generate test scripts where variability is incorporated and specialize them per test case.
- Use (fragments of) formerly generated test scripts to produce test scripts for test cases of new variants

Architecture diagram 1:


Variability model
Variable test script generation
Generic test script
Abstract test case
Test script specification
Variant/test case-specific test script

Architecture diagram 2:


Test case
matching
Test scripts for already instantiated test cases
Test script generation
Test case specific test script

Endpoints:

The program takes the abstract test cases generated by the tool for variant selection above as an input. The output is a test suite of executable test scripts.

Dependencies:
Tool for modelling variability, tool for variant selection

Technology Stack:
Java, Eclipse RCP

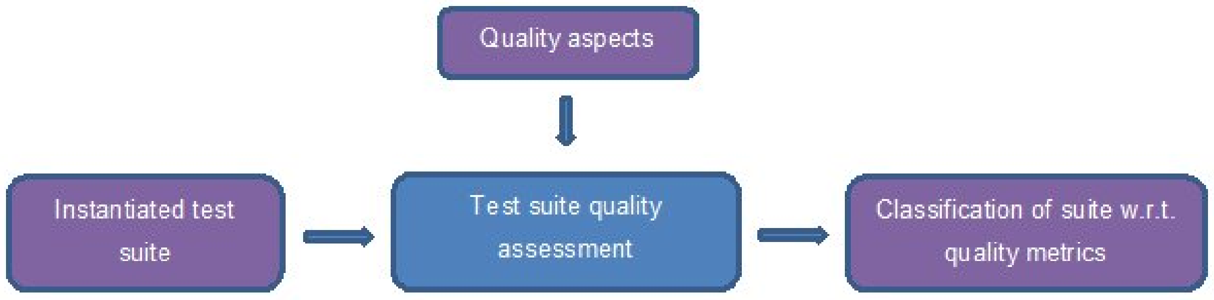
5.3 Service for Test Suite Quality Assessment

Owner: Expleo

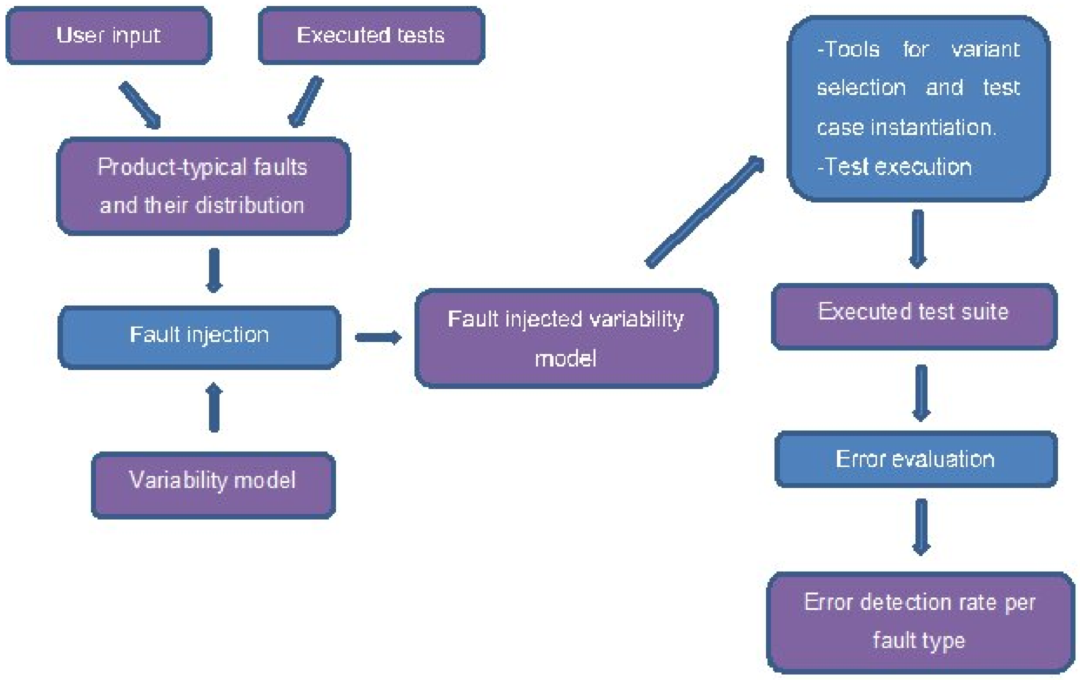
Category:
This tool is part of deliverable D3.3 of T3.2.

Description:
The functionality of the tool comprises the assessment of different quality aspects (e.g. given in IEC/ISO 9126) of the test suite coming out of the tool for variant selection and test case instantiation. In particular, it evaluates the instantiated test suite with regard to error detection capability by fault injection into the base model. The below architecture for injection of system-internal faults could later be expanded to implement security testing via fault attacks.

Architecture diagram (general):


Quality aspects
Instantiated test suite
Test suite quality assessment
Classification of suite w.r.t. quality metrics

Architecture diagram (for fault injection):


User input
Executed tests
Product-typical faults and their distribution
Fault injection
Variability model
Fault injected variability model
-Tools for variant selection and test case instantiation.
-Test execution
Executed test suite
Error evaluation
Error detection rate per fault type

Endpoints:
The program takes either the instantiated test suite and a choice of quality aspects, or the base model and product-typical faults in order to create a fault injected test suite as an input. The output is an assessment of the quality aspects in terms of the metric assigned to the aspect.

Dependencies:
Tool for modelling variability, tool for variant selection, tool for test case instantiation

Technology Stack:
Java, Eclipse RCP, Angular JS, possibly UI of platform

5.4 Service for Test Injection and Execution (FBDMutator)

Owner – MDH

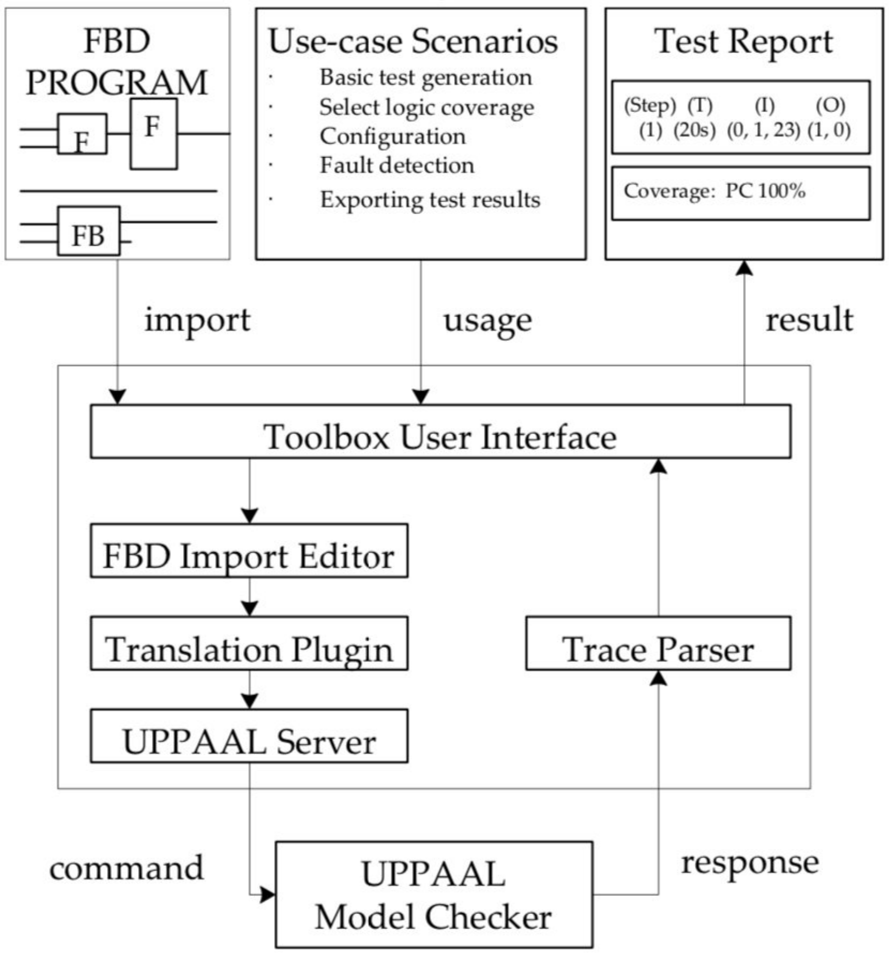
Category – WP3

Description – FBDMutator is a software solution intended to be used for automated testing of Programmable Logic Controllers (PLC) software, found in systems like airplanes, nuclear power plants, medical devices, trains, space shuttles.

PLC software is all around us. We use them in our daily life without even considering to what extent their functioning is dependent on software and that they might fail because of a software. And there is one thing that makes these systems rather unique about them: if they fail, people may die and the environment may be at harm. However, testing these systems is a rather difficult and time consuming task. There are many national and international standards, safety regulation agencies mandating a certain level of testing. Currently, many companies developing PLC software are manually testing their software which is a tedious and error prone process.

FBDMutator is rooted in theories of mutation testing, simulation and model checking, and test execution in simulation.

Architecture diagram



Endpoints
FBD Import Editor. This module is used for validating whether the structure of a provided XML file represents a valid PLCOpenXML file containing an FBD Program.

Output. The function of the interface is to provide a way for the user to communicate with the tool including: (1) the selection of which FBD program to import and execute tests for, (2) the selection mutation operators to be used for test execution and evaluation, (3) the presentation of generated test results, and (4) the determination of correctness of the result produced for each generated test by comparing the mutated test output with the expected output from the original program. Additionally, the tool can export the results in a csv format.

Dependencies

The UPPAAL Server module is used for external invoking of the UPPAAL model checker. UPPAAL provides support for formal verification using a client-server architecture, allowing the toolbox to connect as a client to the model checker and verify properties against the model.

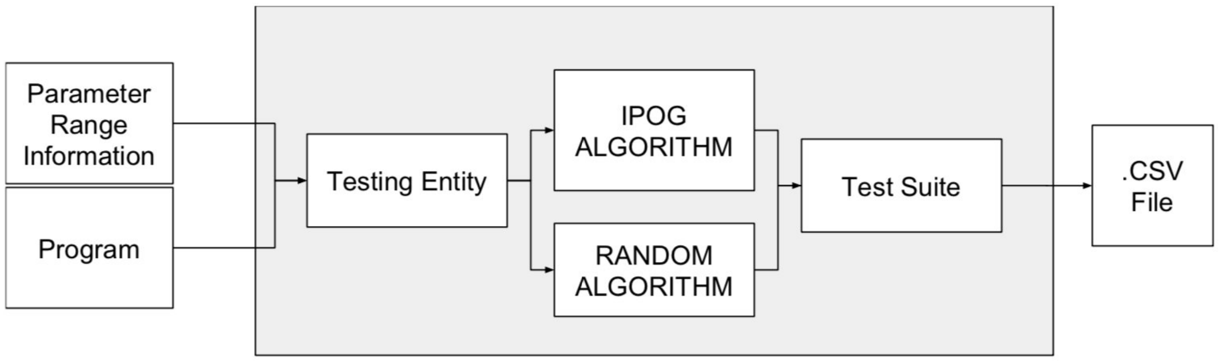
Technology Stack – Java, JavaCC, UPPAAL

5.5 Service SEAFOX

Owner – MDH
Category – WP3
Description – SEAFOX is the only available combinatorial test suite generation and selection tool for industrial IEC 61131-3 control software. SEAFOX is open source software and is available at https://github.com/CharByte/SEAFOX

SEAFOX supports the generation of test suites using pairwise, base choice and random strategies. For pairwise generation, SEAFOX uses the IPOG algorithm as well as a first pick tie-breaker. SEAFOX was used in several studies in order to support testing of industrial programs and fault detection. A tester using SEAFOX can automatically generate test suites needed for a given industrial IEC program after manually providing the input parameter range information based on the defined behaviour written in the specification.

Architecture diagram



Endpoints
FBD Import Editor. This module is used for validating whether the structure of a provided XML file represents a valid PLCOpenXML file containing an FBD Program.

Output. The function of the interface is to provide a way for the user to communicate with the tool including: (1) the selection of which FBD program to import and generate tests for, (2) the selection of the coverage criterion to be used for test generation, (3) the presentation of generated test inputs, and (4) the determination of correctness of the result produced for each generated test by comparing the actual test output with the expected output (as provided manually by the tool user). Additionally, the tool can export the results in a csv format.

Dependencies

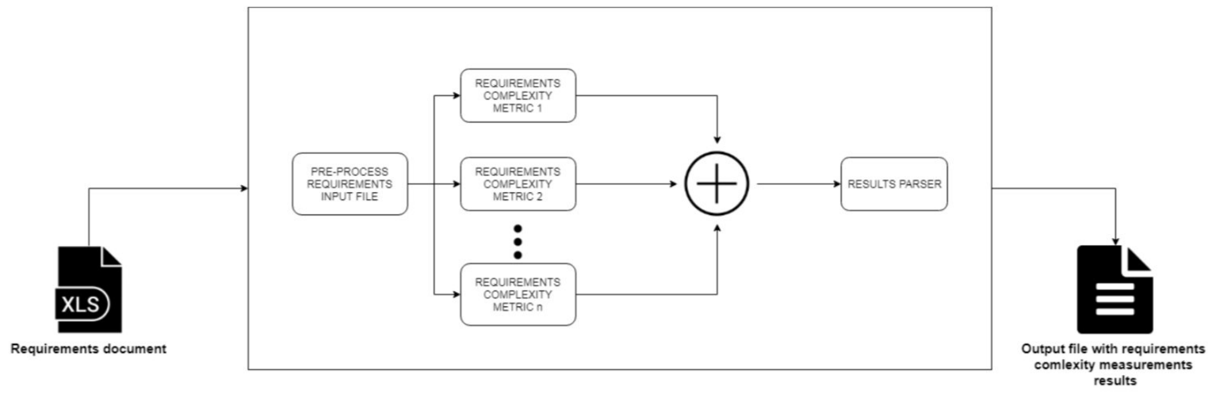
No dependencies

Technology Stack – .NET

5.6. Service RCM (NALABS)

Owner – MDH
Category – WP2
Description – RCM was developed as an effort to create an automatic requirements complexity measurement tool for industrial systems.

Architecture diagram


REQUIREMENTS COMPLEXITY METRIC 1
PRE-PROCESS REQUIREMENTS INPUT FILE
REQUIREMENTS COMPLEXITY METRIC 2
RESULTS PARSER
REQUIREMENTS COMPLEXITY METRIC n
XLS
Requirements document
Output file with requirements comlexity measurements results

Endpoints
Reading Excel documents into the application and parsing the data into a format that is easy to manipulate within C#.

Output. The tool can export the results in a csv format.

Dependencies

No dependencies

Technology Stack – .NET

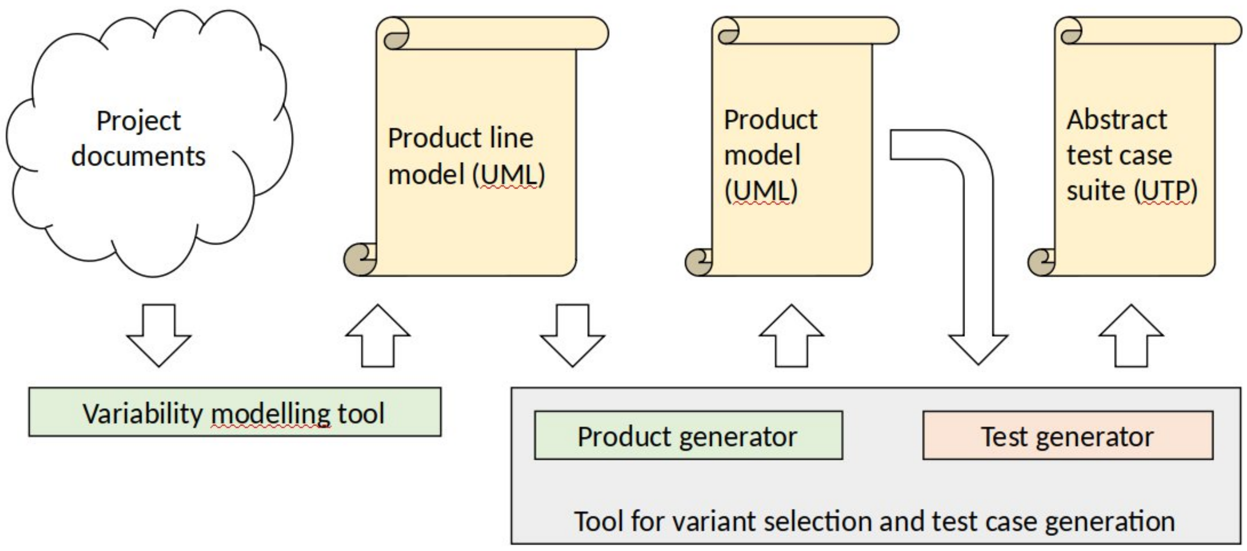
5.7 BeVR: Service for requirements-based variability modelling

Owner: QA Consultants

Category: WP3

Description: The purpose of this tool is to enable a human operator to convert information from project documents into a Product Line Model, which describes a family of systems in terms of shared features and development history. Creation of a Product Line Model enables automatic generation of abstract test cases (T3.1) and efficient variant selection for test optimization (T3.2).

Architecture diagram: The diagram below presents the variability modelling tool in its usage context. Using project documents as reference, a human user creates a Product Line Model (PLM) with the tool's help. This model is encoded as a collection of UML diagrams. These are then passed to the tool for variant selection, which according to its own directives use the PLM to instantiate models for individual products and corresponding abstract test case suites. PLM's and Product Models are encoded as sets of UML diagrams, while test suites follow the UML Testing Profile (UTP) notation.


Project documents
Product line model (UML)
Product model (UML)
Abstract test case suite (UTP)
Variability modelling tool
Product generator
Test generator
Tool for variant selection and test case generation

Endpoints: Tool input will be manually provided by the user through a Graphical User Interface (GUI). As the result of the user's work, a Product Line Model will be generated, in the form of a collection of UML diagrams encoded as XML documents.

Dependencies: None, as the tool stands at the beginning of the test case generation workflow.

Technology Stack: The tool will be based on Papyrus (a modelling tool itself built on top of the Eclipse IDE), with added UML stereotypes, customized UI, convenient install packaging and other optimizations for Product Line Model generation.

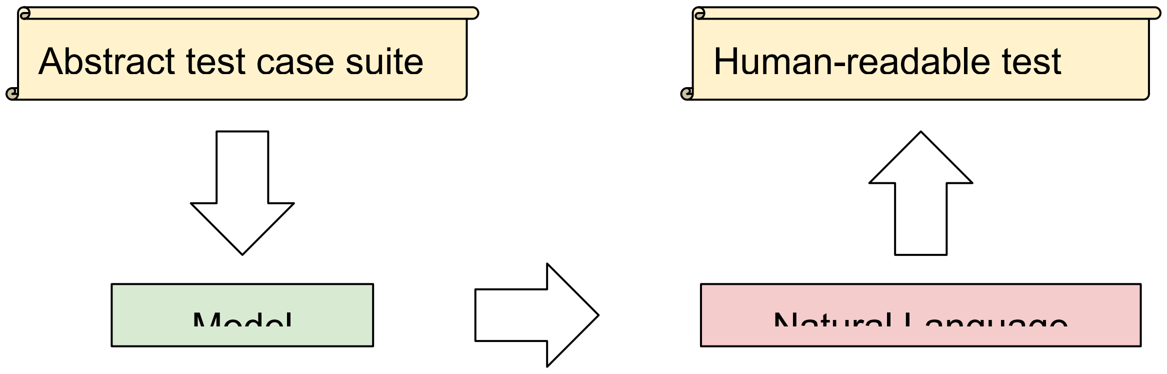
5.8 Service for generating human-readable test scripts

Owner: QA Consultants

Category: WP3

Description: There are many scenarios where automated generation of test cases can help optimize time and resources, yet test execution itself must be done manually. For example, in the automotive domain, testing Advanced Driving Assistance Systems (ADAS) requires actual vehicles and human drivers for proper evaluation and recovery in case of failure, but precisely because the expenses of real-world tests cannot be avoided, optimal test selection is all the more critical. Therefore, a tool that converts abstract test cases into human-readable scripts will be implemented to bridge the gap between automated test generation and manual test execution (T3.3).

Architecture diagram: The tool will be composed of a parser that converts test cases to an internal representation, and a natural language generator that generates human-readable scripts from that representation. See diagram below for an illustration.



Endpoints: The tool will take as input the abstract test cases produced by the tool for variant selection, encoded as UML Test Profile (UTP) diagrams, and output test scripts written in human-readable natural language.

Dependencies: Tool for variant selection.

Technology Stack: The tool will be implemented in Java, using the UML2 library to manipulate the abstract test cases and SimpleNLG for natural language generation.

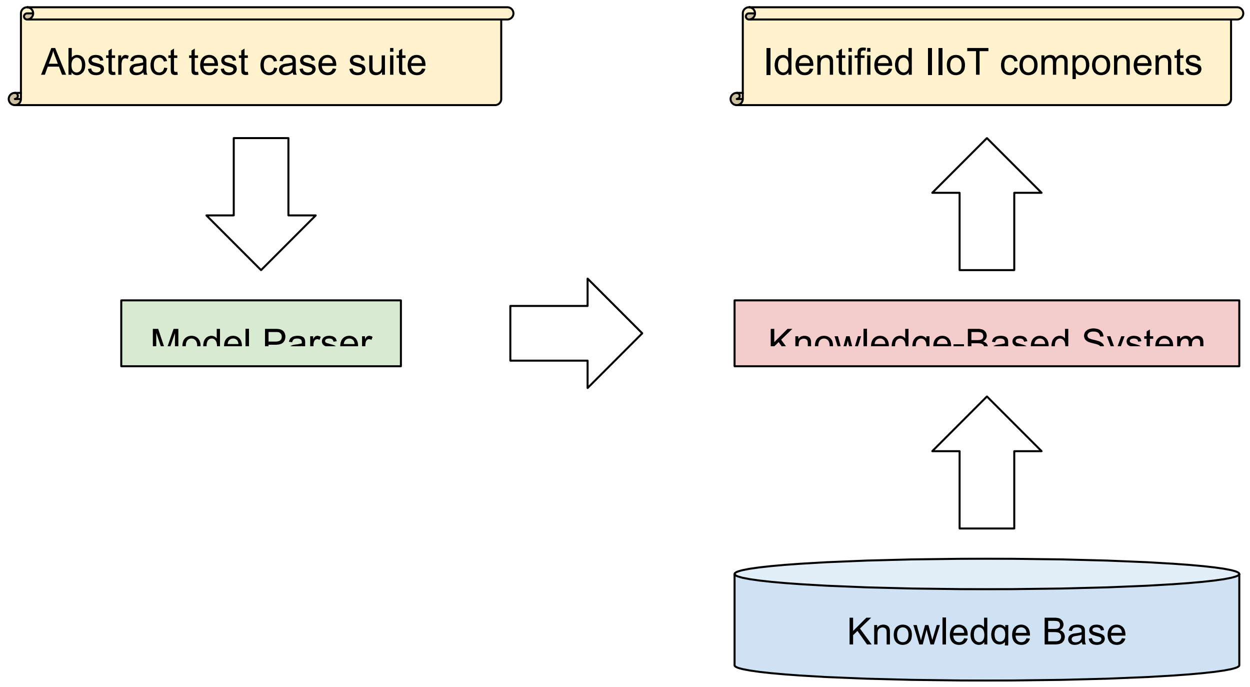
5.9 Service for IIoT component identification

Owner: QA Consultants

Category: WP3

Description: Industrial Internet of Things (IIoT) “refers to interconnected sensors, instruments, and other devices networked together with computers' industrial applications, including manufacturing and energy management. This connectivity allows for data collection, exchange, and analysis, potentially facilitating improvements in productivity and efficiency as well as other economic benefits.” [^] IIoT systems and components are of particular concern to cyber-security; therefore, a tool will be developed to identify IIoT components and interfaces in larger systems, allowing other inspection tools to place extra focus on them (T4.5).

Architecture diagram: The tool will be composed of parser and inference modules. The parser converts test case suites to an internal representation suitable for automatic manipulation. The inference module is a Knowledge-Based System (KBS) that relies on domain-specific knowledge bases for identifying IIoT components.



Endpoints: The tool will take as input the abstract test cases produced by the tool for variant selection and a domain-specific knowledge base, and produce as output a list of identified IIoT components.

Dependencies: Tool for variant selection.

Technology Stack: The tool will be implemented in Java, using the UML2 library to manipulate the abstract test cases and d3web as the basis for the knowledge-based system.

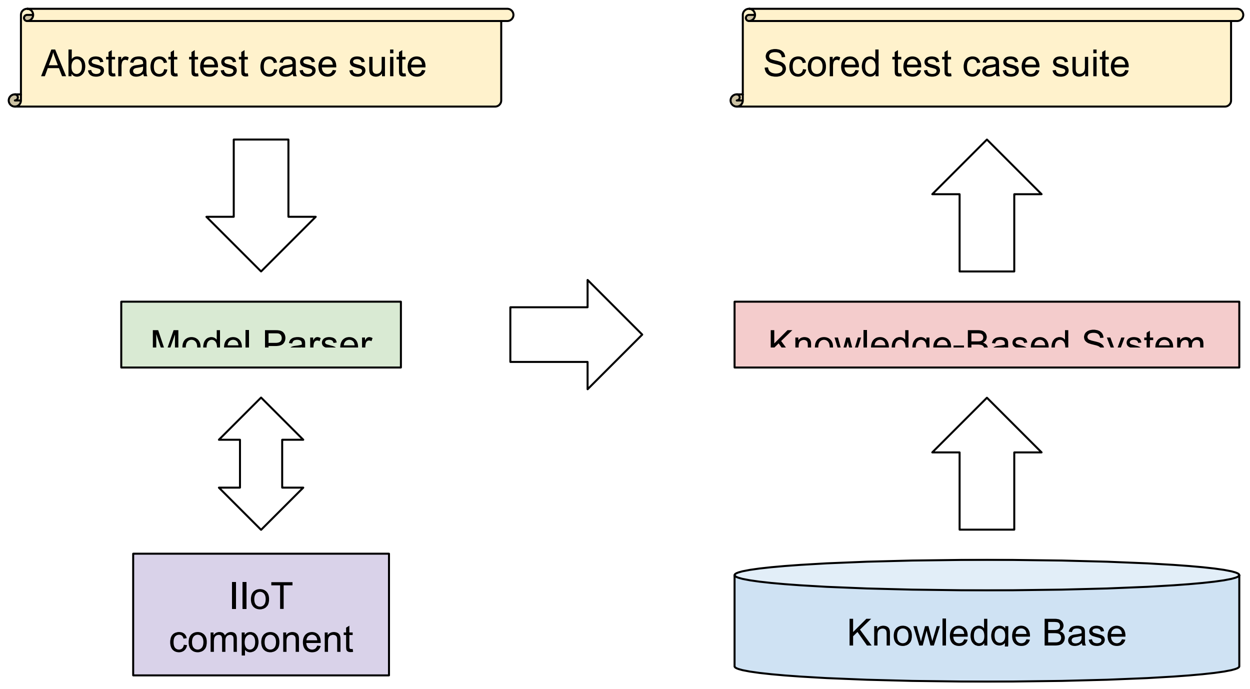
5.10 Service for risk-based test scoring

Owner: QA Consultants

Category: WP3

Description: Safety and security risks are two critical dimensions for optimal test case prioritization. Therefore, a tool will be implemented to search test cases for patterns indicative of such risks, assigning risk scores as pertinent. Such scores can then be used alongside other metrics (e.g. feature relevancy) to appropriately prioritize test cases, or exclusively for choosing candidates for fault and attack injection exercises (T3.4). The tool will be built around a Knowledge-Based System (KBS) modelling common safety / security vulnerabilities and respective risk assessments, which will require domain-specific knowledge bases to be collected.

Architecture diagram: The tool will be divided into two modules, responsible for different steps in the scoring process. First, a parser converts test case suites to an internal representation, using the IIoT component identification tool to flag the presence of such components. Next, a KBS searches that representation for patterns documented in its knowledge base, producing test case scores in response.



Endpoints: The tool will take as input the abstract test cases produced by the tool for variant selection and a domain-specific knowledge base, and produce as output safety and risk scores for each test case.

Dependencies: Tool for variant selection, tool for IIoT component identification.

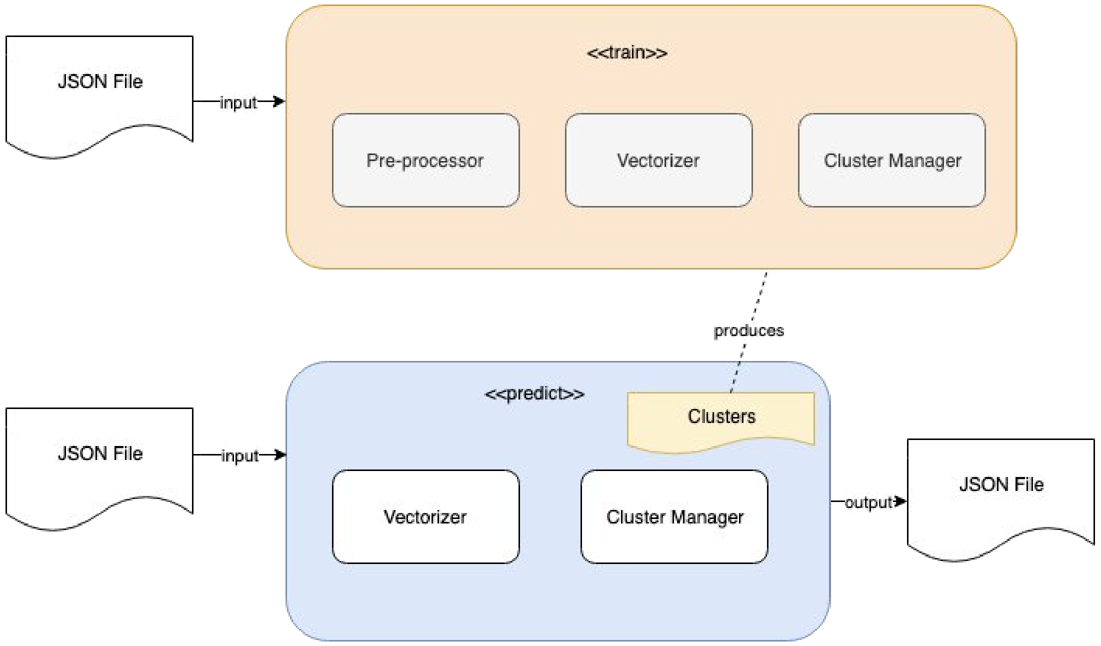
Technology Stack: The tool will be implemented in Java, using the UML2 library to manipulate the abstract test cases and d3web as the basis for the knowledge-based system.

5.11 VARA: Service for requirements similarity-based reuse prediction

Owner – RISE & MDH
Category – This tool is part of WP2 to address the needs of UC2.
Description –
The tool provides two distinct interfaces to the user. *The first interface* will take in the already implemented requirements with links to the reused product line asset's

description. The interface will cluster the requirements based on their semantic similarity while preserving the reuse links. The output of this interface would be a Boolean representing success or failure. *The second interface* takes in a list of key-value pairs. The keys would be IDs and the values will text of the requirements that need to be implemented. The interface will produce a map of the size of input with the requirements IDs on indexes linked with a list of IDs of the product line assets that can be reused to implement the requirement.

Architecture diagram –



Endpoints – Both interfaces 'train' and 'predict' works with JSON input and produces JSON as output. The reason VARA uses train interface is to make it applicable in different domains.

Dependencies – None

Technology Stack – Python

5.12 IntegrationDistiller: Service for abstract integration test case generation

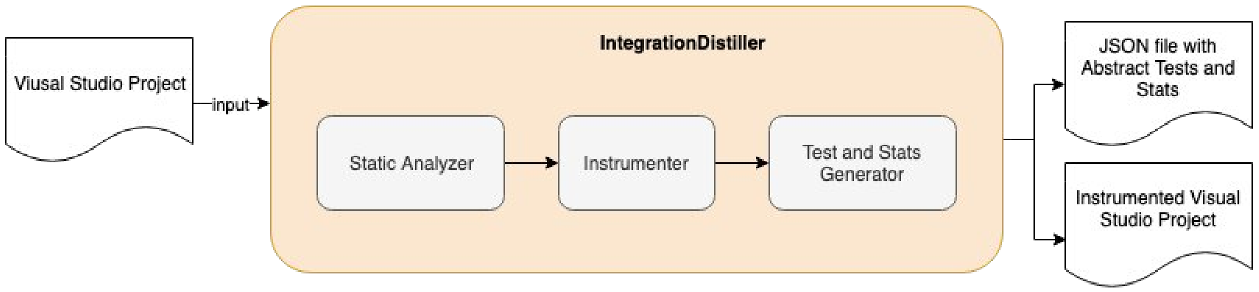
Owner – RISE

Category – This tool is part of WP2 to address the needs of UC3-1.

Description –

IntegrationDistiller utilizes .NET compiler APIs to statically analyze the source code and extract statistical information and integration tree from it. The tool generates integration paths as abstract test cases by traversing the tree paths. In addition, other useful stats like number of unused variables, number of uncalled methods and number of dependent components on a component is also extracted and presented to the user. IntergrationDistller also allows the instrumentation of integration-critical points for timing properties analysis.

Architecture diagram –



Endpoints – Get Visual Studio Projects as input and produce a JSON file and an Instrumented version of the Visual Studio Project
Dependencies – None
Technology Stack – C# , .NET

5.13 SaFReL: Service for performance test case generation

Owner – RISE
Category – This tool is part of WP2.

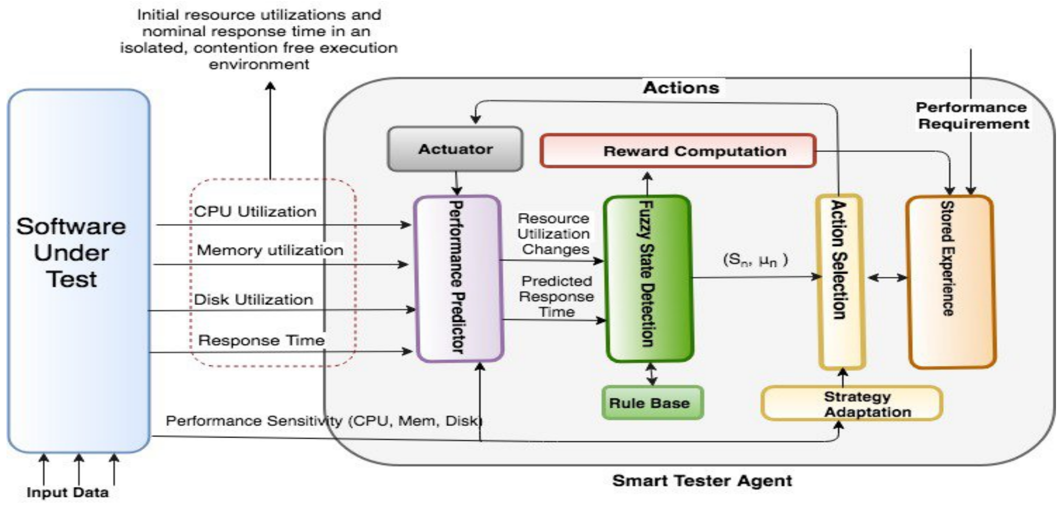
Description –
SaFReL is a self-adaptive fuzzy reinforcement learning-based performance testing framework which makes the tester agent able to learn the optimal policy for generating test cases resulting in performance breaking point without access to model or source code. Finding the performance breaking point of the software under test (SUT), at which the system becomes unresponsive or the performance requirement gets violated, is the intended testing objective in the tool. The current performance testing prototype generates the platform-based test cases by changing the resource availability.

It assumes two learning phases, i.e., initial and transfer learning. First, it learns the optimal policy through the initial learning and then reuses the learnt policy (acquired knowledge) for further SUTs with performance sensitivity similar to already observed ones. It still keeps the learning running in the long-term.

The current prototype uses a performance prediction module to estimate the effects of the applied actions. It gets the initial resource utilization and nominal response time of the system, which have been measured in an isolated, contention free execution environment, and the performance sensitivity indicators as inputs.

This framework could be executed on a virtual machine containing the SUT, and it would be augmented by an actuator doing the resource scaling within the VM. In this case, it will be able to use the (resource) monitoring tools (services) like Percepio Tracealyzer to receive the status data.

Architecture diagram –



Endpoints – Gets the performance data of the SUT and generates the stress test cases
Dependencies – None
Technology Stack – Java

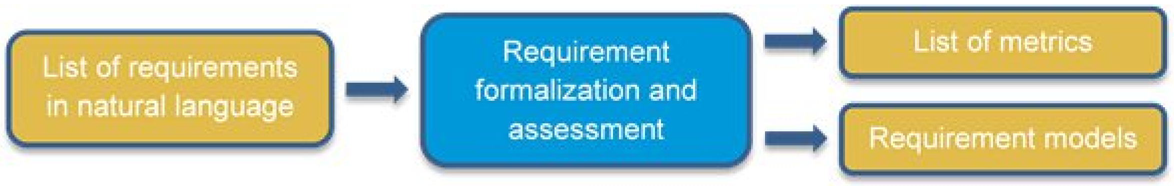
5.14 ReForm: Service for requirement formalization and test optimization

Owner: ifak

Category:
This tool is part of the test object- and feature- based optimization, therefore of T2.3 in WP2, and uses the results of WP3 in regard to variant modeling. It addresses the needs of the use cases UC1 (Automotive, Expleo) and UC3-2 (Industrial Production, FFT).

Description:
This tool will investigate in analysis of textual requirements with knowledge based techniques including natural language processing and machine learning to determine a general risk assessment for variant rich systems. It produces risk based metrics and use case dependent metrics for the purpose of test optimization. Additionally, it will parse the requirements which are written in natural language in order to extract the relevant information and create requirement models.

Architecture diagram:


List of requirements in natural language
Requirement formalization and assessment
List of metrics
Requirement models

Endpoints:

Input:	• a list of informal/semiformal requirements in natural language
Output:	• a list of risk and use case metrics (table of values/percentages)
	• requirement models in IRDL (Ifak Requirements Description Language)

Dependencies:
list of requirements (from use-case providers), tool for test generation and variant traceability (second tool from ifak), possibly UI of platform

Technology Stack:
Python, C++, NLP tools, Machine Learning tools

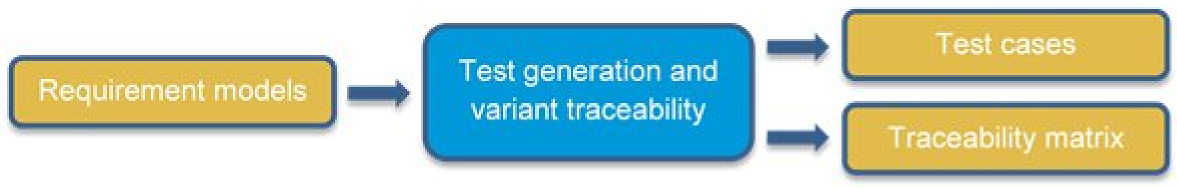
5.15 ifakVBT: Service for test case generation and variant traceability

Owner: ifak

Category:
This tool is part of the test object- and feature- based optimization, therefore of T2.3 in WP2, and uses the results of WP3 in regard to variant modeling. It addresses the needs of the use cases UC1 (Automotive, Expleo) and UC3-2 (Industrial Production, FFT).

Description:
This tool will generate test cases from the requirement models and produce a linkage between the test cases and the corresponding variants of a specific model.

Architecture diagram:



Endpoints:

Input:	• requirement models in IRDL (Ifak Requirements Description Language)
Output:	• generated abstract test cases (XML)
	• a table/matrix of traceability of test cases to variants

Dependencies:
Tool for requirement formalization and test optimization (first tool from ifak), possibly UI of platform

Technology Stack:
Python, C++, test generation tools

5.16 MTest

Owner: MES

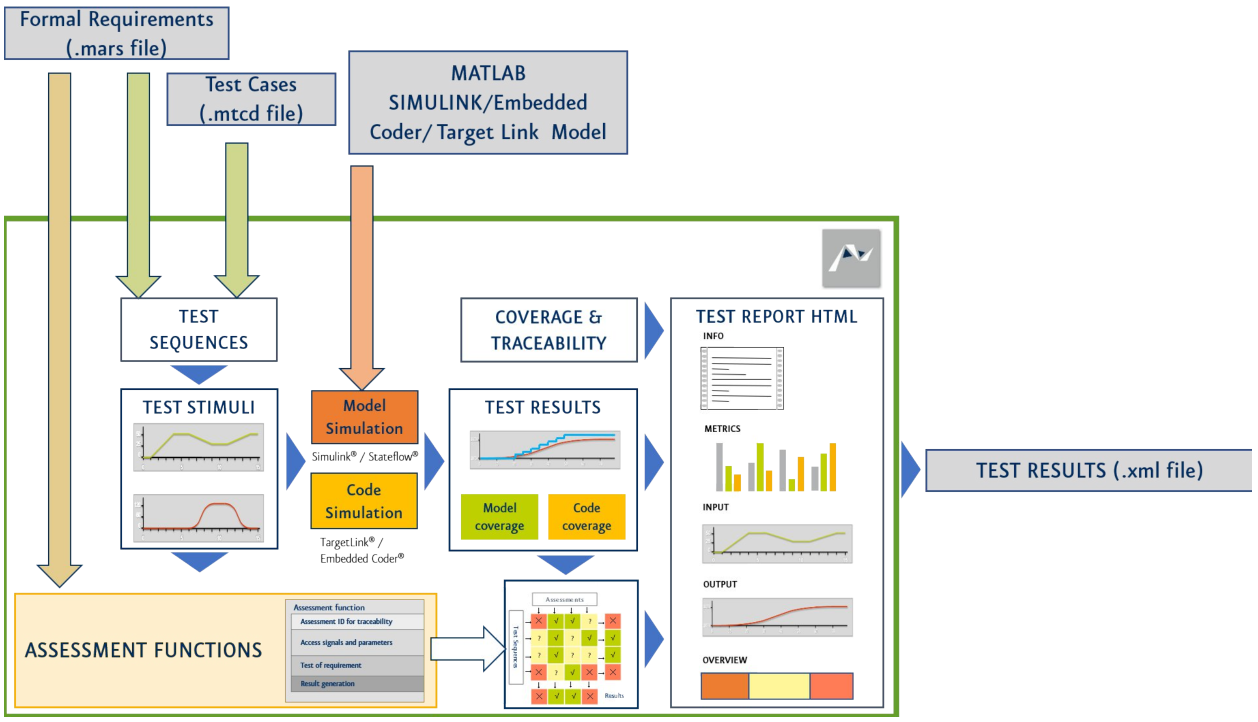
Category: WP3

Description: MES Test Manager® (MTest) is a test management tool that facilitates ISO 26262-compliant, requirements-based testing of Simulink®, Embedded Coder®, and TargetLink® models. MTest automates all test activities for unit and integration testing in terms of functional testing and regression testing. Furthermore, MTest supports all simulation types from model-in-the-loop to processor-in-the-loop simulation so as to support back-to-back testing as well.

MTest guarantees software quality assurance and compliance with standards such as the automotive industry's ISO 26262. In addition, it supports the ISTQB® methodology and techniques. MTest is the first choice for requirements-based testing as it simplifies development, improves quality, and ensures software safety.

Within the XIVT project we develop a test case generator from formalized MARS requirements. Aditionally we inverstigate whether Software Variance can be captured in MARS requirements. Interfaces to the XIVT tool architecture will be provided

Architecture diagram:



Endpoints:

Inputs:

Matlab Simulink, Embedded Coder or Target Link Model

MARS Requirements

MTCD Testcases

Outputs:

XML/HTML Report

Dependencies:

MATLAB SIMULINK 2011 or higher

Target Link 3.4 or higherfor testing Target Link Models

Java 8 or higher

Technology Stack:

Matlab, Java/Xtend, XText, Maven

5.17 RELOAD: Service for test load generation

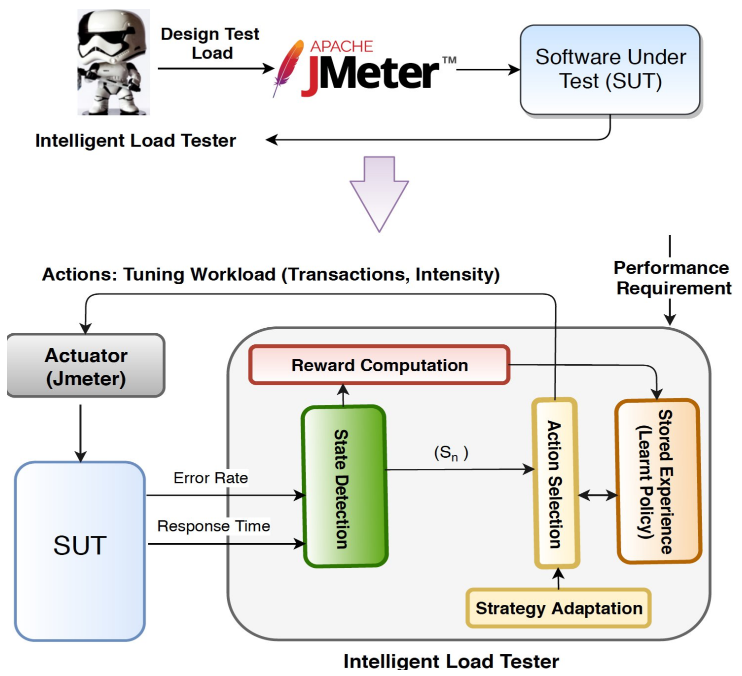
Owner: RISE
Category :This tool is part of WP2.

Description:
RELOAD is an intelligent reinforcement learning-driven load generation tool which generates efficient test load and executes it through a load runner such as Apache JMeter on SUT.

RELOAD learns the optimal policy to generate an efficient test workload which meets testing objective, e.g. reaching an intended error rate, without access to underlying model or source code of SUT. The intelligent tester agent can reuse the learned policy in subsequent potential testing activities such as testing of similar SUTs (software variants) and similar testing scenarios on SUT such as regression load testing.

The learning-based load testing can reach the testing objective with lower cost in terms of workload size (number of users), i.e. smaller workload, compared to a typical load testing process. In summary, generating an efficient test workload meeting intended testing objective, meanwhile, eliminating the dependency on system models and source code are the main strengths of RELOAD as a model-free RL-driven load generation tool.

Architecture diagram



Endpoints: simple .jmx files containing involved requests in running each transaction (operation) of SUT. The jmx files are created by the load runner tool, i.e. Apache Jmeter, through recording an ordinary usage of SUT.
Dependencies: Apache JMeter
Technology Stack: Java

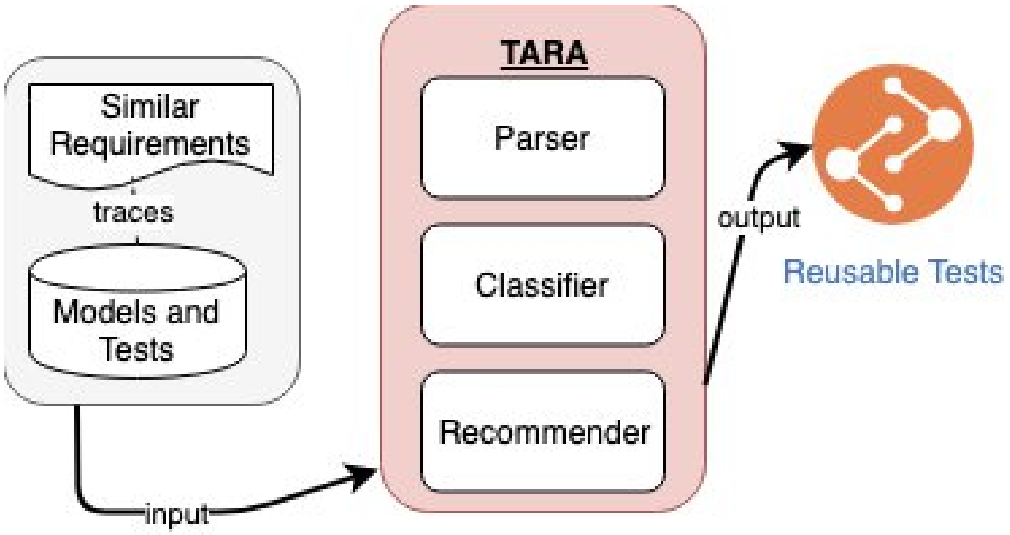
5.18 TARA

Owner: RISE

Category: Test Reuse

Description: Test reuse Analysis and RecommendAtion (TARA) aims to extend our requirements-level similarity engine (VARA) for test effort reduction using horizontal test case reuse. TARA will use the requirements-level similarity among customer requirements and the models realizing them; to identify existing executable test cases that can be reused to test a newly derived product. The test cases might not be directly executable on newly derived products. Thus we aim to develop a semi-automated approach for classification of existing test cases into different classes (such as Reusable as is, Reusable but changes required, and not Reusable).

Architecture diagram:



Endpoints:
Inputs: Similar Requirements and their Models with Test Harnesses
Processing Steps: Test Classification
Output: Reusable Test Cases

Dependencies: VARA, MATLAB/Simulink

Technology Stack: Python, MATLAB/Simulink

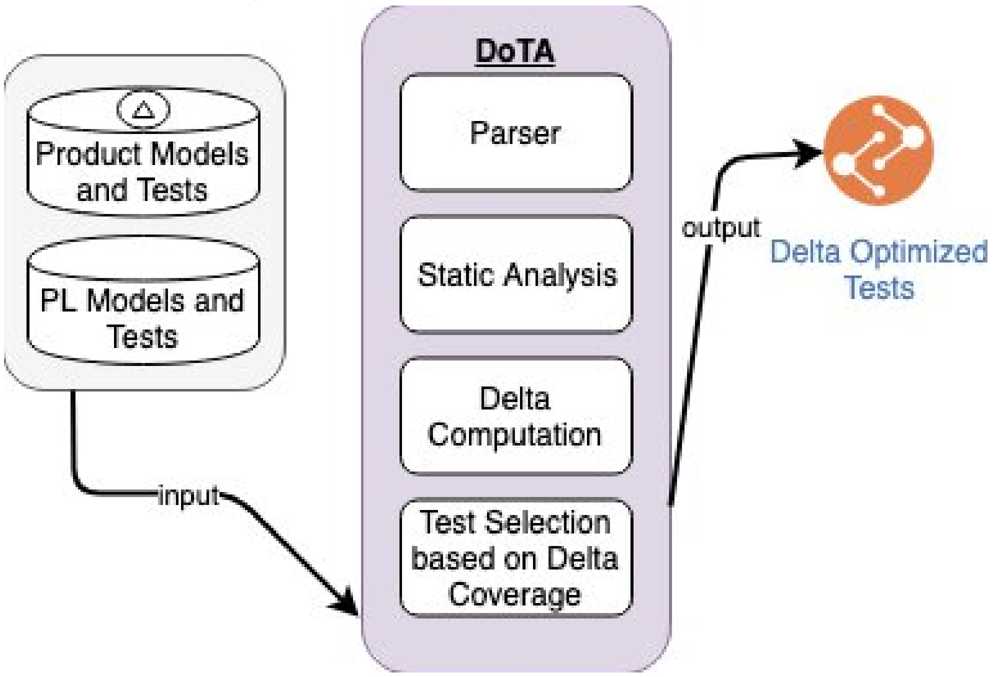
5.19 DoTA

Owner: RISE

Category: Test Priorization, Test Selection, Coverage Analysis

Description: Delta-oriented Test Analysis (DoTA) implements a delta-aware method for test case optimization aided by delta analysis and clone detection. The method will aim to select a subset of executable test cases that provide high coverage to the delta in the product. The method will work with product line (PL) models, PL tests, product models, and product tests (if any) to detect the variable parts of the product from its standard product line and to optimize test cases for high coverage of the variable part of the product.

Architecture diagram:



Endpoints:
Inputs: PL Models with Tests and Product Models with Tests
Processing Steps: Delta Computation, Test Selection based on Coverage to Delta
Output: Subset of Tests (Optimized for Delta Coverage**)**

Dependencies: MATLAB/Simulink

Technology Stack: MATLAB/Simulink

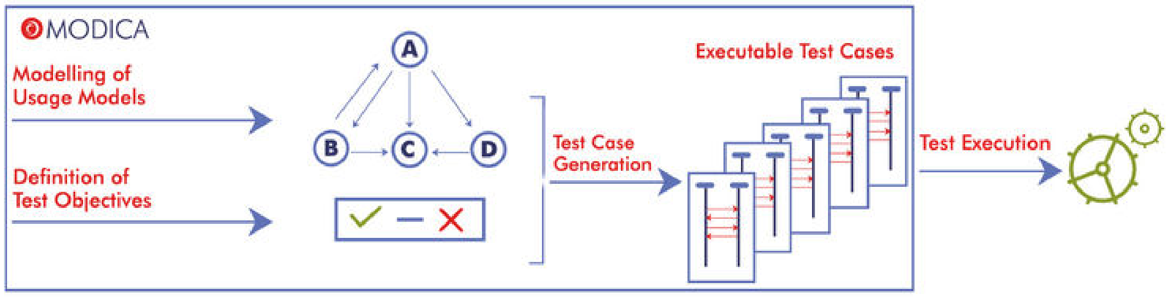
5.20 MODICA

Owner: Expleo

Category:

Description: MODICA is a test generation tool that employs a usage model as a source. To this model, the test case generation algorithm can be applied that aims to comply with specified coverage criteria, using the smallest possible number of test steps in the test cases. Coverage criteria can be given by requirements, the request that (certain) states, state transitions or paths are covered, or the choice of special test sequences that are otherwise hard to reach. In MODICA, there is also a variant handling available that allows to specify test generation strategies for different variants of the usage model.

Architecture diagram:



Endpoints:
Input: Requirements can be imported, e.g. from DOORS.
Output: HTML, PDF, export to EXAM, MESSINA...

Dependencies: See endpoints

Technology Stack: Java, Eclipse RCP

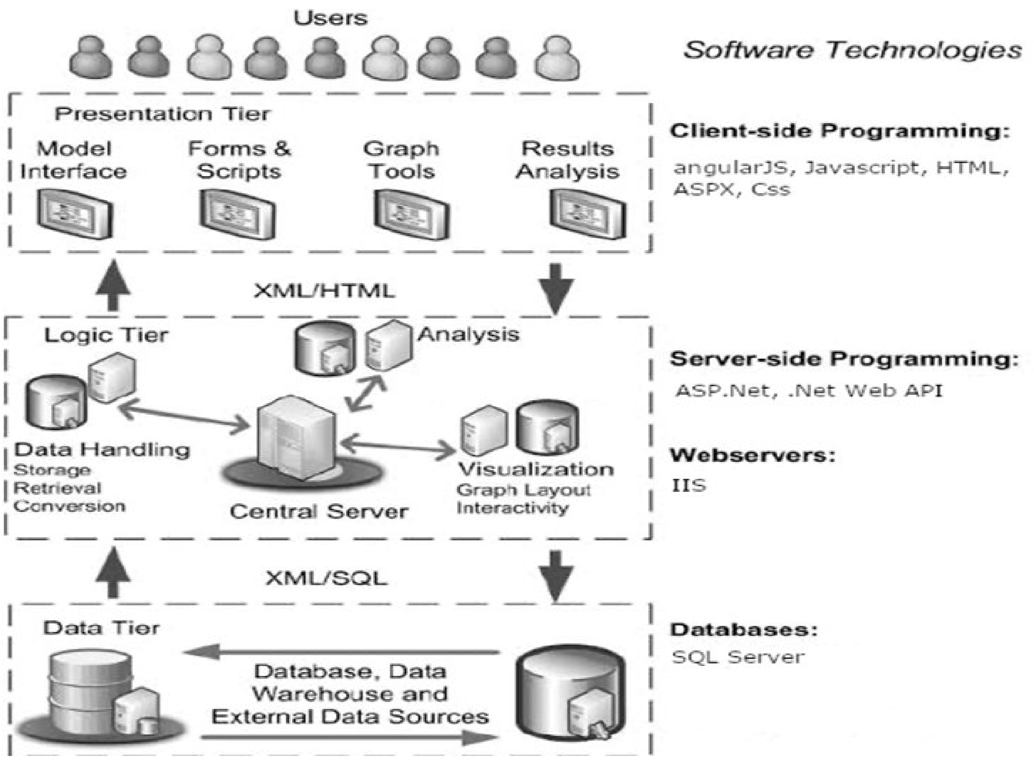
5.21 ARD Test Prioritization Tool

Owner: ARD

Category: Test Modelling, Test Prioritization, Test Selection and Test Logging

Description: ARD is developing a web-based application for the use of test team members. Current version has specialized on testing end-user software products and their variants. It will enable users to define test cases, their priorities and test execution, and allow them to take the logs of execution steps to provide adaptive data for Decision Support System module. In the current state, the requirements and user interfaces of this application have already been defined and the implementation has also been started.

Architecture diagram: The architecture containts Web & DB server(s) for both application and learning data and containts three logical tiers which are presentation, logic/business and data. Three-tier architecture is used for production and development environments by modularizing the user interface, business logic, and data storage layers. This added and important flexibility will improve overall time-to-market and decrease development cycle times by giving development teams the ability to replace or upgrade independent tiers without affecting the other parts of the system. Integration flexibility advantage of tiered architecture is used for embedding analytics parts of the project.



Endpoints:

Input: Any format can be imported.
Output: Depends of input data

Dependencies: No dependencies

Technology Stack: ASP .NET, .NET WEB API, SQL Server, IIS, angularJS, Javascript , HTML, ASPX, Css.

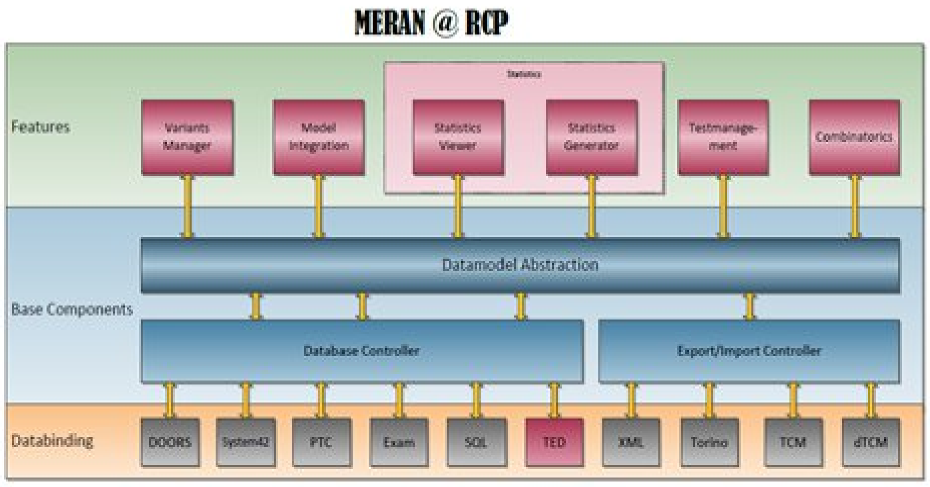
5.22 MERAN

Owner: Expleo

Category: Requirements management

Description: MERAN is an integration tool for requirement management that also supports variant management. It allows the creation of generic entities of requirements or test specifiations, in a way that their properties are fragmented in small units. Once a specific variant is chosen, the requirements or test specifications can be adapted by choice of parameters or text segments.

Architecture diagram:



Endpoints: Data models of the corresponding adapters

Dependencies: Adapters to Doors, Test42, Jira, dTCM,...

Technology Stack: Java, Eclipse RCP, different web services for differences adapters

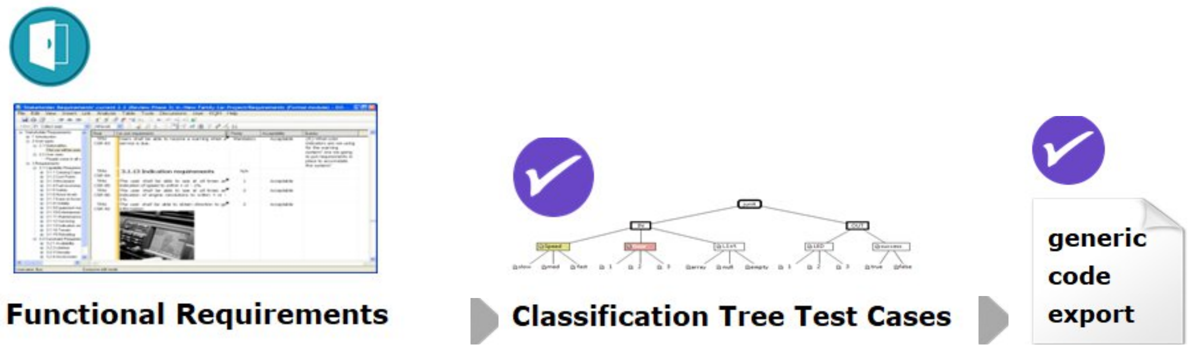
5.23 TESTONA

Owner: Expleo

Category: Modelling, Test case generation, Coverage analysis

Description:
Tool for systematic test design in black-box-tests. All standard specification-based test methods are supported and represented in classification trees.
For the generation of a suite of test cases, there are different modes available that represent different levels of combinatorial coverage. In addition, it is possible to weight the classes depending on their frequency or error risk and consequently obtain a prioritization of test cases. A variability management is built in, allowing the user to specify variants from the generic model and apply TESTONA-applications specifically to them.

Architecture diagram:



Endpoints:
Input: Interfaces to Autosar, DOORS, Matlab, hpALM...
Output: Testona XML files, Interfaces to Matlab, MESSINA, Excel, Word...

Dependencies: See endpoints

Technology Stack: Java, Eclipse RCP

5.24 InnSpect

Owner: Innowave (WinTrust)

Category: Test Execution, Test Adaptation, Test Logging

Description:
InnSpect was developed to address a lack in the testing tools market: one single tool to automate test cases across different devices and different platforms. There are many tools in the market but each one is usually focused in one single technology. InnSpect can start a customer journey using a Web Portal, validate data through API Testing or Database validation, take actions over a Desktop App and finish the journey validating data inside a mobile App.

Architecture diagram: Not Available at the moment

Endpoints: Not applicable

Dependencies: Not applicable

Technology Stack: C#

5.25 DeltaFuzzer

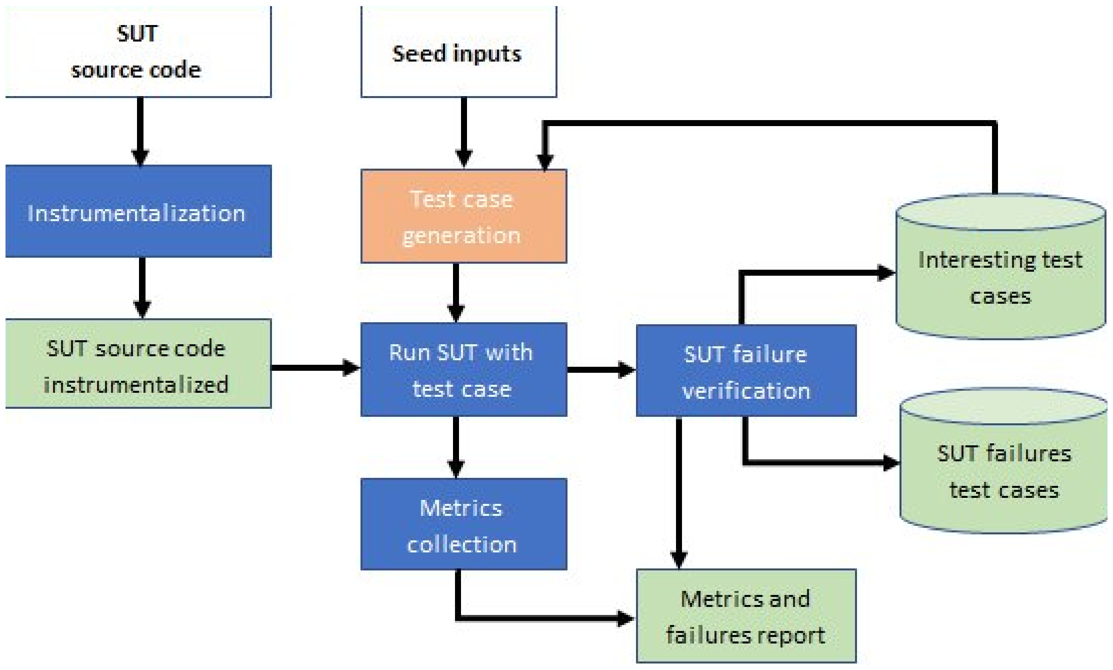
Owner: FCUL

Category: This tool belongs to WP3, being part of deliverable D3.4 of T3.4

Description:

DeltaFuzzer is a grey box fuzzer based on the AFL fuzzer to detect several classes of vulnerabilities presented in software constructed in for C/C++. It is the first fuzzer that implements a ***Targeted Fuzzer Approach*** that makes the fuzzer focus on the (novel) parts that needed to be tested and reuses knowledge acquired in previous testing campaigns. DeltaFuzzer generates a testcase (randomly or through a mutation strategy of existing testcases) for running it in the software under test (SUT) and collects various metrics. Next, it determines if the program suffered a failure, saving thus the test case, and if the test case is "interesting", i.e., if it is capable of uncovering new execution paths and causing a SUT failure, saving it and reusing it to generate another test case.

Architecture diagram:



Endpoints:
Software developed in C/C++ programming languages to be deployed in product variants created by use-case partners. Such software can be the whole program or test cases extracted from the program. The output of the tool is failures and the test cases that caused the failures.

Dependencies:
AFL tool and the source code of programs developed in C/C++ that will be under test.

Technology Stack: LLVM, C, gcc/clang, and python

5.26 TV RoboTester

Owner: Arcelik

Category: Test execution (based on system model)

Description: TV RoboTester is our test automation software developed in house for TV tests. We are able to create test scenarios on the tool dynamically. The tool has its own scripting language. Some features available on the tool are:
1. Actions to simulate test environment
1.1. Simulate user interaction on TV (remote control that can be controlled via PC)
1.2. Play recorded tv broadcast on Stream Players
1.3. etc.
2. Test oracle: Decision points to decide if test result is PASS / FAIL
2.1. Picture capture (LVDS): Live image on TV can be transferred to PC to compare with the reference image
2.2. Picture capture-windows: Live image on TV can be transferred to PC to compare with the reference image (only a portion of the full image)
2.3. OCR: Image from TV is parsed into text and compared to the reference text

Architecture diagram:
Architectural diagram unavailable.
INPUT: Image list of TV software UI
OUTPUT: System Model of TV software with necessary user interactions
We will modify our automation software to work on this system model so that when the model changes our testcases will remain the same.

Endpoints:
Inputs: Test scenarios written manually inside automation software.
+ (After our works within XIVT project) we will use image list of TV software UI (to create system model)
Output: Test results.

Dependencies: No dependencies

Technology Stack: C# for main automation software. C/C++ for device communication frameworks

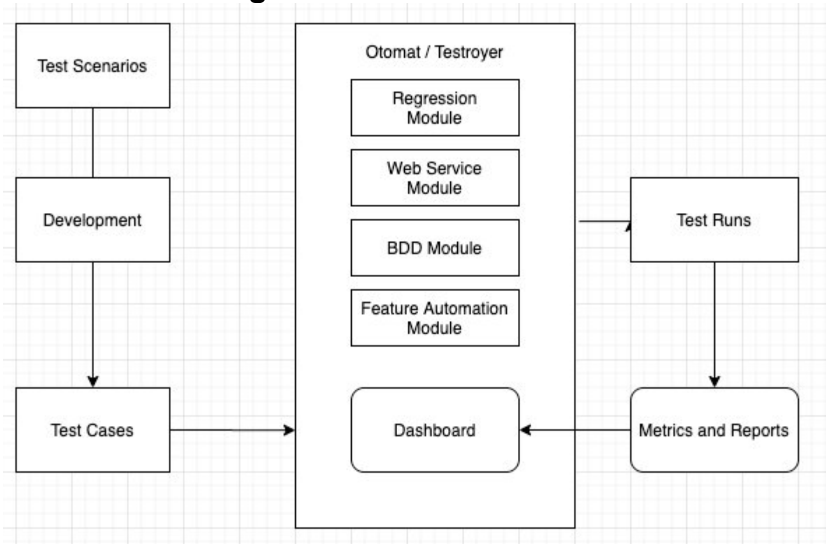
5.27 Otomat (Testroyer)

Owner: Turkcell

Category:

Description: Otomat (Testroyer is new version of otomat with new name and features) is a test management tool. Test Automation engineers take test cases in various forms such as UI, Web Service, BDD and features. Develop required code and store source code in GIT. After OTOMAT triggers daily runs and report test results, categorize them, measure coverage,and send notifications to related parties about test results. BDD and Web Service modules allows users to create their own tests from existing code base manually. In version 2.0 Testroyer we are also adding test selection and prioritization in to our tool. In addition to that we are also planning to add Speech To Test module in our BDD module to our tool.

Architecture Diagram:



Endpoints:
Inputs: Test scenarios with dedicated format for BDD and WS modules, Test scenarios given with natural language developed into a code for Regression module.

Output: Test results, categorization of failures, coverage reports.

Dependencies: No dependencies

Technology Stack: Java, TestNG, Selenium for test case development. AngularJS for frontend, Spring Boot for backend. GIT and Jenkins for test case storage and runs. XRAY for storing user scenarios.

6. Technology Stack Recommendation

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.

6.1 Open-source Framework Suggestions

ModelBus®

On the official website (https://www.modelbus.org), ModelBus® is described as follows:

ModelBus® is a framework for managing complex development processes and integrating heterogeneous tools. It allows to integrate tools from different vendors serving different purposes. This integration creates a virtual bus-like tool environment, where data can be seamlessly exchanged between tools. This avoids the manual export and import of tool specific data, which is usually accompanied by manually executed data alignment steps. The data can be linked by establishing traceability. ModelBus® interoperable tool integration contributes to the collaboration of engineers and developers involved in the software and system development process. (The virtual bus architecture leverages *information exchange between tools and developers.) Thus, it supports tcoordinated simultaneous work. ModelBus® automation is the key to increase the efficiency in a software and system development environment. ModelBus® facilitates the automatic and semi-automatic execution of process steps throughout the complete software development process.*

The key concept of ModelBus® for tool interoperability is the virtual bus-like service-oriented architecture and the way it processes the data transmitted via this bus. ModelBus® can work on traditional artifacts like source code or binaries, but its full potential lies in the handling of models. Tool data can be transmitted via ModelBus® as well-defined MOF/EMF based models, which enables the full power of model-driven engineering practices to the ModelBus® data management. This includes the application of model-transformation techniques, consistency checks and full traceability across multiple process steps ranging from requirements to code for example.

Due to that approach every piece of information created during the development process is accessible and usable for the process and its control. Tools connected to ModelBus® can offer or consume services acting on these data. In that way functionality – provided by individual tools – becomes available for the whole development process and can be used in automated process steps.

ModelBus® is applicable in various domains including embedded systems design, IT-Business, automotive and avionics. The ModelBus® framework makes it possible to create flexible development solutions adapted to the customer's needs. New tool adapters can be built upon request. It shows its full benefit in medium or large development processes but ModelBus® can be used for small solutions as well. Using ModelBus® will help to improve performance of the development and test processes by injecting automation to the highest possible degree. ModelBus® helps to keep the existing processes and tools unchanged. Therefore it helps to save licensing costs and training of developers.

The basic set of ModelBus® is open source and free software. Tool adapters, consultancy, support and maintenance services are available for establishing a ModelBus® based development scenario fitting to individual needs.

Open Services for Lifecycle Collaboration

On the official website (https://open-services.net), OSLC is described as follows:

The OSLC Core Specification is a Hypermedia API standard currently mainly adopted in software and systems engineering domains, but with the potential to provide value to any domain with data integration challenges. The OSLC Core specifications expands on the W3C LDP capabilities, to define the essential and common technical elements of OSLC *domain specifications and offers guidance on common concerns for creating, updating, retrieving, and linking to lifecycle resources.*

OSLC domain-specific specifications define the equivalent of schemas in RDF for enabling data interoperability. They consist of RDF vocabularies and OSLC resource shapes. RDF vocabularies are used to describe standardized resource types and properties. OSLC resource shapes are used to define constraints such as multiplicity constraints on properties of specific resource types.

Moreover, the following value propositions are stated:

As a tool vendor, you need to ensure that your customers can integrate your product with other tools in order to extract the most value from your product. While providing a REST API is a norm nowadays, a developer has to build an integration layer. As every REST API is different, it means extra time reading the documentation, extra time developing plumbing code to perform model transformation, and most importantly, all this needs to be done on a case basis, leading to point-to-point integrations.

OSLC allows you to provide:

- *a uniform self-descriptive REST API;*
- *a linked data model based on standard domains, common in ALM/PLM (RM, QM, CCM, etc.), that you can tailor to your product;*
- *exchange data in plain JSON with the clients that are not linked-data ready;*
- *provide rich UIs from your tool for use in 3rd-party tools for seamless linked data workflow;*
- *and many other features that your customers would appreciate.*

A number of other products used in ALM/PLM already implement OSLC and your OSLC-enabled tool can integrate with many of them without extra development effort.

As a tool buyer, you have a unique set of requirements towards your toolchain and for many reasons (technical, organisational, financial) a single-vendor solution might not be viable for you. Therefore, one of the most important criteria for procurement of the new software tools is their TCO including the integration costs. Most of the tools come with semi-open proprietary APIs that often lack documentation. Those APIs will incur considerable development costs, often involving highly specialised consultants with a deep knowledge of the tool in question.

Tools that come with an OSLC-based API will you to integrate them into your toolchain with less (or none, in many cases) development effort, while performing a deeper integration, at the workflow level.

OSLC reduces the complexity and risk of increasingly complex software infrastructures, and improves the value of software across a broader set of internal and external stakeholders. OSLC-based API is an experience truly free from a vendor lock-in.

As a tool user, you have to switch between a plethora of tools on a daily basis. You often don't see how their integration is done, but you feel that it's done poorly: updates showing up in other tools many hours later, integrations getting broken every other tool update, etc.

Well-implemented OSLC integrations mostly remain backwards-compatible even across major releases. Standardised OSLC APIs often allow vendors to provide a fully supported integration with many other OSLC-compliant tools out of the box. Finally, an OSLC integration can be performed not only at the level of two tool data models, but at the level of your workflow involving those tools. This is possible through the use of delegated UIs, which allow you to interact with another OSLC-compatible tool without leaving your current open tool!