ITEA3

ASSUME

Affordable Safe & Secure Mobility Evolution

Interoperability Criteria

Deliverable D3.3

Deliverable Information			
Nature	Report	**Dissemination Level**	Public
Project	ASSUME	**Project Number**	14014
Deliverable ID	D3.3	**Date**	April 7th, 2017
Status	Final	**Version**	V1
Contact Person	Jan Steffen Becker	**Organisation**	OFFIS
Phone	+49 441 9722 529	**E-Mail**	jan.steffen.becker@offis.de

Author Table

Name	Company	Email
Jan Steffen Becker	OFFIS	jan.steffen.becker@offis.de
Philipp Reinkemeier	OFFIS	philipp.reinkemeier@offis.de
Björn Lisper	MDH	bjorn.lisper@mdh.se
Reinhold Heckmann	AbsInt	heckmann@absint.com
Matthias Kern	FZI	mkern@fzi.de
Stefan Otten	FZI	otten@fzi.de
Matthias von Steimker	Berner & Mattner	Matthias.von-Steimker@berner-mattner.com
Frédéric Loiret	KTH	loiret@kth.se
Udo Brockmeyer	BTC	udo.brockmeyer@btc-es.de
Philippe Baufreton	SAFRAN	philippe.baufreton@safrangroup.com
Bernard Schmidt	BOSCH	Bernard.Schmidt@de.bosch.com
Heiko Dörr	MES	doerr@model-engineers.com
Frank Benders	TNO	Frank.Benders@tno.nl
Staffan Nyström	FindOut	staffan.nystrom@find-out.se

Change and Revision History

Version	Date	Reason for Change	Affected pages
V1-draft1	January 2, 2016	Initial version	1-13
V1-draft2	Feb 2, 2017	Added description of SWEET	Section 3.3
V1-draft3	Feb 3, 2017	Added sections on AbsInt analyzers	Sections 3.4 and 4.2
V1-draft4	Feb 6, 2017	Added section on MES Quality commander	Sections 3.5 and 4.3
V1-draft5	Feb 10, 2017	Added section on BTC EmbeddedPlatform	Sections 3.10 and 4.6
V1-draft6	Feb 14, 2017	Added sections from SAFRAN and FindOut	Sections 2.3 and 3.8
V1-draft7	March 1, 2017	First complete draft	Sections 4 and 5
V1-draft8	March 23, 2017	Added section on timing analysis provided by OFFIS	Section 3.2
V1-draft9	March 23, 2017	Added section on detection of high level race conditions	Section 2.6
V1	April 7, 2017	Final version	Sections 1 and 5

Table of Contents

AUTHOR TABLE 2

CHANGE AND REVISION HISTORY 3

TABLE OF CONTENTS 4

LIST OF FIGURES 6

1. EXECUTIVE SUMMARY 7

1.1 GLOSSARY OF TERMS 7

2. NEEDS OF END-USERS 10

2.1 KIT AND FZI: TRACEABILITY FOR STATIC CODE ANALYSIS 10

2.2 BERNER&MATTNER BER_UC01 11

2.3 SAFRAN ELECTRONICS & DEFENSE 12

2.3.1. Model based design applications 13

2.3.2. Hand coded applications 14

2.4 TNO TNO_UC01_4 14

2.5 CORNERING LIGHT DEMONSTRATOR AS PART OF THE BODY CONTROLLER USE-CASE BY DAIMLER AG 15

2.6 DAIMLER AG: DETECTION OF HIGH LEVEL RACE CONDITIONS 16

2.7 ROBERT BOSCH GMBH 18

3. TECHNOLOGY PROVIDER CAPABILITIES 19

3.1 OFFIS CONSISTENCY ANALYSIS 19

3.2 OFFIS SYSTEM-LEVEL TIMING ANALYSIS 20

3.2.1. Interface to concurrency defect analyses 20

3.3 SWEET – SWEDISH EXECUTION TIME TOOL 21

3.4 ABSINT – STATIC PROGRAM ANALYSIS 22

3.4.1. aiT 22

3.4.2. Astrée 23

3.5 MES QUALITY COMMANDER FOR ONLINE QUALITY MONITORING 23

3.6 KTH – OSLC / LINKED DATA ADAPTORS DEVELOPMENT KIT 24

3.7 KIT AND FZI: TRACEABILITY FOR STATIC CODE ANALYSIS 25

3.8 FINDOUT TECHNOLOGIES AB – VISUALIZATION OF LINKED DATA RESOURCES 25

3.9 B&M CODE SECURITY ANALYSIS 26

3.10 BTC-ES: MBT AND FORMAL METHODS SUPPORT 26

4. BENEFITS OF ENHANCED INTEROPERABILITY 29

4.1 OFFIS CONSISTENCY ANALYSIS 29

4.2 ABSINT – STATIC PROGRAM ANALYSIS 29

4.2.1. Combined System and Code Level Timing Analysis 29

4.2.2. Static Analysis in Model-Based Software Development 29

4.2.3. Static Analysis and Model-Based Testing 30
4.3 MES QUALITY COMMANDER FOR ONLINE QUALITY MONITORING 30
4.4 KIT AND FZI: TRACEABILITY FOR STATIC CODE ANALYSIS 31
4.5 CODE SECURITY ANALYSIS 31
4.6 BTC-ES: MBT AND FORMAL METHODS SUPPORT 31
5. SYNTHESIS OF INTEROPERABILITY BENEFITS 32
5.1 INTEROPERABILITY SCENARIOS 32
5.1.1. KIT and FZI: Traceability for Static Code Analysis 33
5.2 EXCHANGE FORMATS 35
5.2.1. XML Timing Cookies (XTC) 35
5.2.2. Comparison to ASSUME SCA tool exchange formats 36
6. CONCLUSIONS AND DISCUSSION 37
REFERENCES 38

List of Figures

FIGURE 1: DIRTY READ/UNCOMMITTED DEPENDENCY 16
FIGURE 2: NON-REPEATABLE READ 17
FIGURE 3: LOST UPDATE 17
FIGURE 4: TASK NETWORK EXAMPLE 20
FIGURE 5: KTH OSLC / LINKED DATA ADAPTORS DEVELOPMENT KIT 25
FIGURE 6: AS1 - INTEROPERABILITY CONCEPT WITH AN IDE, A STATIC CODE ANALYSIS TOOL AND A DESIGN AND ARCHITECTURE TOOL ON THE CLIENT SIDE 34
FIGURE 7: AS2 - INTEROPERABILITY CONCEPT WITH AN IDE AND A DESIGN AND ARCHITECTURE TOOL ON THE CLIENT SIDE AND A STATIC CODE ANALYSIS TOOL ON THE SERVER SIDE 34

1. Executive Summary

The goal of this deliverable is to identify needs for interoperability from both, the perspective of end-users of methods and tools, and the perspective of technology providers. This results in a view of topics about interoperability that should be worked on during the ASSUME project. Further, in this deliverable also needs for interoperability are identified that may go beyond the scope of the ASSUME project, giving rise to an agenda for pursuing interoperability topics beyond the ASSUME project.
This deliverable is structured as follows. Chapter 2 collects end-user's needs and present workflows in the ASSUME project that may benefit from improved interoperability. Chapter 3 lists the methods and tools that are available within the ASSUME consortium and describes capabilities and interfaces of the tools. Chapter 4 describes how these tools can benefit from interoperability, e.g. improved applicability or accuracy of a method if interoperability with another tool is provided. Chapter 5 concludes by merging the information from the previous chapters and identifying synergies among use cases and tools. Additionally, the existing XTC format for timing analysis and the evolving OSLC (Open Services for Lifecycle Cooperation) based interoperability approach by FZI and KIT are described here. In Chapter 5, also a first analysis of the required and provided data and requirements of the tools and use cases is performed that may be a first step towards standardization.

1.1 Glossary of terms and abbreviations

Term	Definition
Consistency	A set of requirements is consistent if there is at least one implementation of the system that fulfills all requirements, i.e. the requirements do not contradict each other.
Formal requirement	A requirement written in formal language with fully defined semantics.
Pattern	A pattern is a template for a formal requirement with parameters that are filled in by the engineer with concrete expressions.
Semi-formal requirement	A requirement written in formal language but without fully defined semantics.
Software Component	A software component is a software element with well-defined interfaces that can be independently deployed and composed with other software components.
ECU	An Electronic Control Unit (ECU) is a term typically used in the automotive domain and

Term	Definition
	refers to an embedded device that controls one or more electrical systems or subsystems in a vehicle.
Operating System Task	A task is a particular function that is executed by an operating system and is subject to the scheduling strategy applied by it.

Abbreviation	Definition
ASIL	Automotive safety integrity level
MBT	Model-based testing
DAL	Design assurance level
DMA	Direct memory access
ECU	Electronic control unit
GUI	Graphical user interface
HIL	Hardware-in-the-loop
IDE	Integrated development environment
KPI	Key performance indicator
MCP	Multi-core processor
MIL	Model-in-the-loop
MPPA	Massive parallel processor array
NoC	Network-on-chip
OEM	Original equipment manufacturer
OIL	Open image library
PIL	Processor-in-the-loop
QoS	Quality-of-service
SCA	Static code analysis
SCP	Single-core processor

Abbreviation	Definition
SIL	Software-in-the-loop
WCET	Worst-case execution time
WCTT	Worst-case traversal time
WP	Work package
XTC	XML timing cookie

2. Needs of end-users

This chapter collects needs from end-users regarding methods supporting particular design step(s) in a development process. This includes all kinds of analysis of properties of a design artifact, like absence of deadlocks, fulfillment of a requirement, proving refinement of requirements, synthesis steps computing new design artifacts and/or parameterization of existing ones based on a given set of requirements.
The needed method is described from a methodological perspective and possibly also a technical perspective. That means a technical perspective further refines the needed method by adding required technical realization details. Based on such identified method, needs for interoperability are derived.

2.1 KIT and FZI: Traceability for Static Code Analysis

In this section, the requirements for static code analysis tools and the overall methodology and framework in order to achieve traceability between development artifacts are described.

In view of safety and security, the traceability between all safety relevant design artifacts shall be possible in order to provide rationale for the complete safety case. Thus the ISO26262:2011 claim for example “Safety requirements shall be traceable…” ([1], 6.4.3.2) or “The traceability of safety-related hardware elements shall be ensured,…”([1], 7.4.5.3). Furthermore, the traceability between the design, the implementation, the validation and test results should be possible to ensure in every point in time that all data are valid. In context of static code analysis, traceability should be achieved between the analysis results of the tool and the corresponding software components, model artifacts such as Matlab/Simulink or Enterprise Architect and requirements.

The analysis results from different analysis tools should be comparable, to use the different strength and capabilities from different tools. A traceability and interoperability of different static code analysis tools would enable higher quality of analysis results and analysis depth.

World-wide collaboration with valid and consistent data during the different development phases of automotive software systems should be possible. Due to growing internationalization of development teams, collaboration is a key factor in future system development. Additionally, development processes and methods are heterogeneous across different vehicle domains due to different goals and constraints. The targeted approach should consider these constraints appropriately.

There are different static code analysis tools on the market, which differ in strength and weaknesses as well in their capabilities. Thus, an application scenario, where a combination of different static analysis tools are used is reasonable in order to get superior quality of the software. This is currently difficult to do, because there is no standardized interchange format for static code analysis tools. Actual there is a need for two interchange formats. One for the analysis results and one for the configuration, which describes the target system.

2.2 Berner&Mattner BER_UC01

The Berner&Mattner use case BER_UC01 is centered around projects that encompass the recognition of lane and traffic signs, sensor fusion for classification and localization of obstacles, automated parking maneuvers, collaboration between cars, cooperative security (e.g. during evasion). The main objective is to provide a development platform for new car technologies like autonomous driving, Car2Car and Car2X communication.

Embedded software engineering has successfully addressed many well understood software quality criteria like correctness, or robustness. So far, however, automotive security as one key quality criterion has been nearly unaddressed over the last decades in automotive engineering, even though several research studies already showed the risks associated with insecure automotive software design. The range of threats rising from security weaknesses spans from illegal access to car, privacy violations over manipulation of less critical car features to the manipulation of safety relevant functions like brakes, steering system, or powertrain. Today, there seems to be no comprehensive approach for systematically designing software security into the system nor are there comprehensive and systematic approaches for validating automotive security aspects. From an end-user point of view in the BER_UC01 use case, a methodology and/or automated tools for validating automotive security aspects is requested.

Today's automotive systems usually make use of many different technologies. Hence, a tool realizing a security analysis method shall cover very different concepts and specific attack scenarios for these technologies and programming languages. As a result, we expect a more comprehensive, reproducible security analysis yielding comparable results among different projects.

To effectively and efficiently detect relevant security vulnerabilities, the following interoperability criteria between B&M Code Security Analysis and supporting analysis tools must be fulfilled.

TecReq_BM_01: Support for Programming Languages and Compilers for Embedded CPUs Commonly Used in Automotive Industry
Commonly used programming languages as well as compilers have to be supported in order to guarantee optimal benefits in productive use. The programming languages C (up to Version C99) and C++ (at least up to Version C++11) must be supported. The parser shall be user-configurable to allow adaptions, e.g., to compilers which are not directly supported. Predefined configurations for the most commonly used compilers for embedded CPUs should be available out of the box.

Many different compilers are used for embedded systems in the automotive area. In order for flexible usage the tools shall parse source code for a wide range of these systems. Predefined configurations for the most common compilers reduce the effort for the setup process.

TecReq_BM_02: Robustness
When parsing various code bases many syntactical peculiarities can occur. The tool(s) shall be able to handle parse errors by giving detailed information to the user (e.g., by logging information and through outputs in the interface) and proceed the parsing process when errors are less severe, i.e., recoverable.

When more serious errors occur, the tool(s) shall mark a returning point, i.e., a point in the process before the error occurred and which allows the user to start the processing from there, instead of starting from the beginning. This shall allow the user to be able to manually correct problematic parts of the code.

Also the turnaround time to retry the parsing after correction of a reported error shall be as low as possible. This should be achieved by the possibility to reuse successful parts of a past parse run.

TecReq_BM_03: Mapping between Results of Analysis Iterations and Different Tool Versions
The tools shall be backward compatible, i.e., able to handle outputs created by older tool versions for comparison of results and trend analyses.

TecReq_BM_04: Practicable and Predictable Runtime
The tool(s) shall have a fast runtime that makes them practicable to work with industrial code bases for electronic control units (ECUs).

It must be possible to retrieve a reliable runtime prediction for the parsing and analysis process to be able to make credible assumptions about the total run-time. The predicted run-time of the current task, as well as the overall run-time shall be displayed to the user. The user shall always have knowledge about how long the tool needs to perform the current active task and the expected duration for the next task, as well for the complete run.

TecReq_BM_05: Interoperability
The developed tools shall be interoperable, i.e., provide accessible interfaces to a broad range of data and control sources and sinks. The data exchange shall be executed via well defined, documented and open interfaces.

2.3 SAFRAN Electronics & Defense

Certification authorities and applicants are concerned that the interference between software applications executing on an MCP (Multi-Core Processors) could cause safety-critical software applications to behave in a non-deterministic or unsafe manner, or could prevent them from having sufficient time to complete the execution of their safety-critical functionality. However, MCPs were designed to provide a substantial increase in performance over traditional single-core processors (SCPs). Having several cores integrated onto one device could allow several functions to be integrated together on one processor and in one piece of equipment. Such a scenario is foreseen in use case SAF_UC2 (description in D1.1) which is composed of SAF_UC1.x (DAL A) for the hard real time control part running on one dedicated Bostan MPPA® cluster and several not critical Heath Monitoring (HM) functions (DAL D/E) running on other clusters with QoS constraints (guaranteed services).

EASA requests applicants for installations involving the use of MCPs in safety-critical systems to meet the objectives provided in the CRI MCP. In the frame of the ASSUME context, SAFRAN Electronics & Defense will focus on software development and verification e.g. architectural guidelines, development effort from traditional model-based applications running on single core to several cores (KPIs defined in D1.2), degree of automation, fulfillment of verification objectives through static analyzers. A major expectation is the availability of a trustable semi-automatic tool-based so-called triangle where edges are respectively model/code (Use case 1.x/use case 2), executable code and corresponding verification activities. Such feedback loop in the triangle should offer the possibility to quickly refine user annotations whenever appropriate and perform the activities along the triangle again to reach a better mapping, improve efficiency, performances, memory consumption etc. Optimization of any given tasks / response time / allocation is not in scope of ASSUME. However DAL A objectives should be met and the correct-by-construction principle remains for SAF_UC1.x.

From a methodological perspective, the system developer provides a functional specification of the software and a non-functional specification that contains timing, deadlines and the description of the hardware architecture which is most often selected before the project start. Software design and development have the purpose to map the elements of the functional specification on the hardware architecture model (checking hardware compatibility), and generate code corresponding to the result of this mapping step. One key aspect is that the produced code will run while preserving the semantics of the functional specification, and satisfy the requirements described in the non-functional specification. Hard real time systems require maximum predictability so that strong guarantees have to be met and checked.

Avionic system software is written in a high level modeling language e.g. Simulink/SCADE for the synchronous applications (Use cases 1.x) and in C for other types of software such as Heath Monitoring Systems (Use case 2). We will briefly describe the needed method from a technical perspective for both hereafter.

2.3.1. Model based design applications

As described in SAF_UC01 - synchronous application, UC1.2 is a 16 cycles (period) DAL A scheduler based on Simulink / SCADE-kcg that actually runs on a single core processor. For the purpose of the use case, the functional part e.g. dataflow has been removed and replaced by fake functions thus keeping traceability with timing specifications.

The need is to be able to apply a sound and verifiable methodology (correct-by-construction) for Synchronous Data Flow programs written in Simulink/SCADE and mapped them to multiple parallel dependent tasks running on one compute cluster of the Kalray MPPA® Bostan many-core processor. This includes synchronization of the communication resources, memory mapping etc. The use cases SAF_UC1.x will be analyzed with respect to real-time implementation and means to reduce as much as possible the effort on the MPPA® integration while preserving the correct behavior. Performance aspects are not of interest in that use case but the mapping onto 16 cores while ensuring temporal and spatial partitioning with respect to EASA CRI MCP objectives.

A first technical view of what the use cases 1.2 (and 1.3 later on) require in terms of data and tools includes SCADE (and Heptagon) designs with non-functional requirements (real-time, partitioning, allocation...), Lopht / SCADE-MPPA® in order to generate MPPA® source code, configuration files and mapping. Target compiler (CompCert) to produce executable per core might be used.

Regarding verification activities, the intra-cluster configurations will be checked through some of WP2/WP5 tools from Absint and possibly others if necessary. The types of verification should include intra cluster memory coherency checking (cache purged & invalided at right instants…), intra clusters thread synchronization checking, inter cluster synchronization checking, DMA inter clusters, DMA cluster to DDR checking (address ranges, capacity, timings…). Other types of analysis also include run-time errors analysis, worst-case execution-time (WCET), worse-case traversal time (WCTT). As a first step, tool(s) that analyze(s) the generated code and give a predicted WCET for the parallelized application will be used (all executed cores) then end-to-end guarantees will be considered.

From a usability perspective, interoperability is seen as an enabler to smoothly operate verification in line with DAL A objectives before and after loading application(s) on the MPPA. With regards to the type of analysis to be done, adaptability/flexibility is expected from a user perspective by access to configuration items, hardware modelling of the cluster, Network-on-Chip (NoC) of the Bostan MPPA® and constraints (if necessary) that must be respected by the runtime in order to develop/assess formal tools using this hardware model and guidelines to avoid bottlenecks or possibly unsafe behavior. Adaptability/flexibility is important for further mapping when iterations will be required to improve timing execution reaching some optimality at run-time.
Tools interoperability between synthesis of real-time parallel code with formal guarantees of functional and non-functional correctness, covering both the synthesis algorithms themselves and the modeling of the execution platform would save significant effort and reduce error prone interfaces data handling between tools.

Standardized interfaces to establish the interoperability with the individual analysis tools, but also with requirements and design tools might help/ease the smooth operation of tools that will be used in the ASSUME DAL A toolset. A potential integration between development and verification tools (academic and commercial) should contribute to the ramp up of the multi/many cores processors maturity.

2.3.2. Hand coded applications

The main difference from a methodological point of view is that the design/code is done manually, hence verification tools/activities described above should be complemented by code checking (coding rules, boundaries, pointers usage, etc).

2.4 TNO TNO_UC01_4

The TNO use case TNO_UC1_4 is related to traceability of requirements. The main objective is to create and extend the existing TNO tool set such that it shall support the traceability to the design, implementation, testing of functionality and safety.

In the TNO organization the requirements of the system and the safety process for automotive applications are maintained in Enterprise Architect (and sometimes exported to Microsoft Excel). In Enterprise Architect tool all requirements are stored in a structured format. The global high-level design is made in Enterprise Architect while the detailed design and implementation are performed in Matlab/Simulink. The verification is first performed with a test framework in Matlab/Simulink followed by Hardware-In-the-Loop testing and operational validation tests on vehicle(s). The test plans, descriptions and reports are documented in Microsoft Word. The test documents can be filled using input that is specified in Enterprise Architect (using an export function) and test performed with Matlab Simulink.

In ASSUME also some other tools are developed to analysis designs and perform timing analysis with POOSL (Parallel Object-Oriented Specification Language). These tools should also be linked to Enterprise Architect to generate input for the analysis.

The goal of TNO use-case is to define a way of working and use a tool chain such that all deliverables/artifacts of the systems and safety engineering process are linked and the traceability can easily be performed.

The interoperability requirements are:

- Interface between requirements and design in Enterprise Architect and implementation in Matlab/Simulink.
- Interface between Test Scripts in Matlab Simulink and requirements in Enterprise Architect.
- Interface between design/architecture in Enterprise Architect and design analysis tooling (e.g. fault tree analysis).
- Interface between design/architecture in Enterprise Architect and timing analysis in POOSL.

2.5 Cornering Light Demonstrator as Part of the Body Controller Use-Case

The body controller module (BCM) is an electronic control unit responsible for monitoring and controlling various electronic functionalities in a vehicles body. This section focusses on the cornering light demonstrator as part of the BCM. In general the functionality is highly distributed on various ECUs (Electronic Control Units) that communicate via the car's vehicle busses, such as CAN, LIN, and Automotive-Ethernet. The demonstrator simplifies this as the network communication is abstracted. The textual requirements and software implementation are in focus.

- Requirements are available in natural language
- The implementation is done using the model-based development tools Simulink and TargetLink where Simulink in used for modelling the functionality and TargetLink is used to generate ANSI-C source code.

During the workflow, the following is needed:

- A formal requirement specification language that allows specifying functional requirements and verification procedures, where both test and analysis procedures can be derived from.
- Functional requirements that are checked against the implementation using different technologies such as model in the loop (MIL), software in the loop (SIL), static analysis, or tests. How a functional requirement is checked depends on the complexity of this requirement and the specific needs of the use case provider.
- Traceability of safety requirements on system level and implementation level is needed. A forward analysis shall identify potential impacts of a defect.

Currently, the input formats to specify functional requirements and verification procedures differ significantly between tools and used technologies. Therefore it is expensive or even practically impossible to move a check procedure for one requirement from one tool or technology to another even if this would safe effort in the verification phase. To reduce these costs it is necessary to have an interoperable toolchain for formalization of functional requirements, formal verification, evaluation and model quality.

It is expected from an interoperable toolchain that the effort required to set up and employ an analysis is reduced. Efficiency gains for requirements engineering and testing in the development process are achieved.

2.6 Daimler AG: Detection of high level race conditions

Parallel software requires a proper synchronization of parallel execution paths and shared data/resources. Missing or not sufficient synchronization leads to undefined/inconsistent states of data/resources. The following are examples for high level race conditions resulting from improper synchronization, which shall be detected by analysis tools:

a. **Dirty Read/Uncommitted Dependency:** Data/resource is updated by one thread and read by another in parallel, but the update is not completed yet, so the reading thread will read inconsistent data/resource.

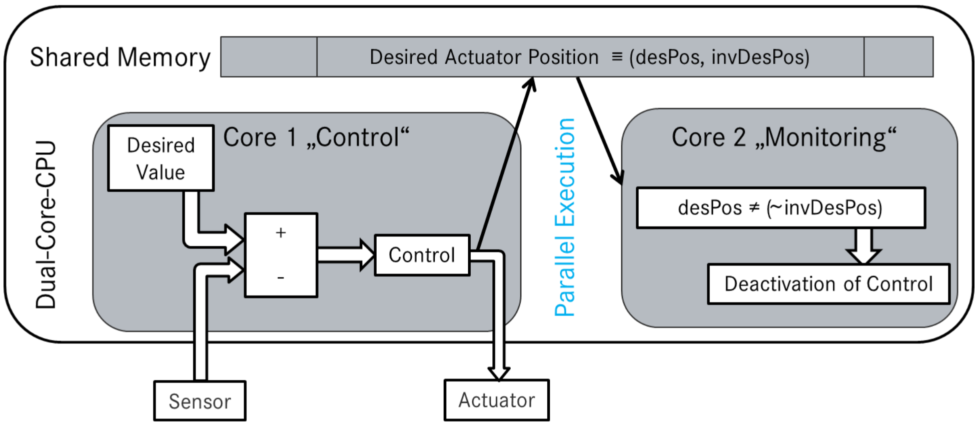
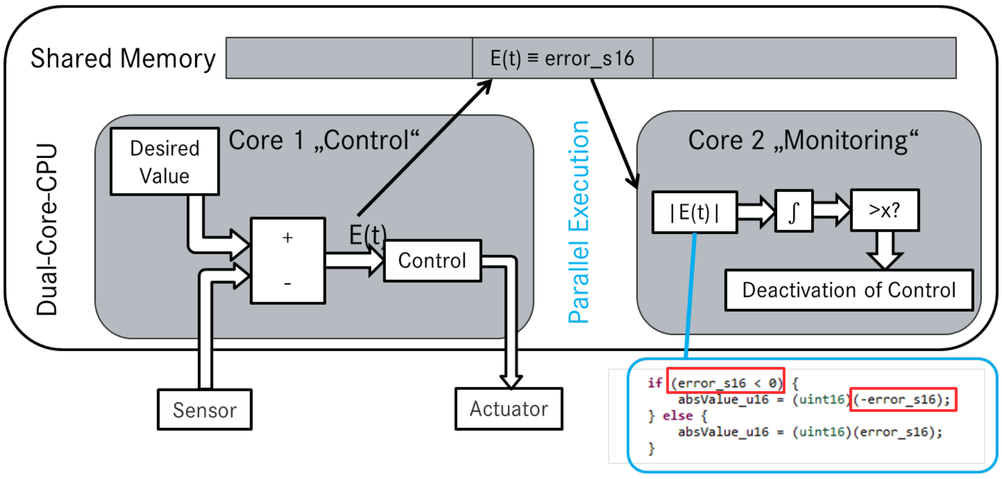


Figure 1: Dirty read/uncommitted dependency

In the example shown in Figure 1, the desired actuator position is stored as normal and inverted value (to detect memory corruption in ASIL software). If the monitoring functionality is reading both values during the update of these values by the control functionality, an inconsistent read (inverted value does not fit to normal value) is very likely.

b. **Non-repeatable Read/Inconsistent Analysis:** One thread reads data or resources multiple times. Every time the information is read, the values have changed because of continuous updates by a second thread.



```
if (error_s16 < 0) {
    absValue_u16 = (uint16)(-error_s16);
} else {
    absValue_u16 = (uint16)(error_s16);
}
```

Figure 2: Non-repeatable read

In the example shown in Figure 2, an update of `error_s16` between 1st and 2nd line of code can lead to an overflow and finally to an unintended deactivation of the control function (safety measure).

c. **Lost/Buried Update**: Two threads write/update the same resource. The update of the second thread will overwrite the update of the first one. The first information is lost.

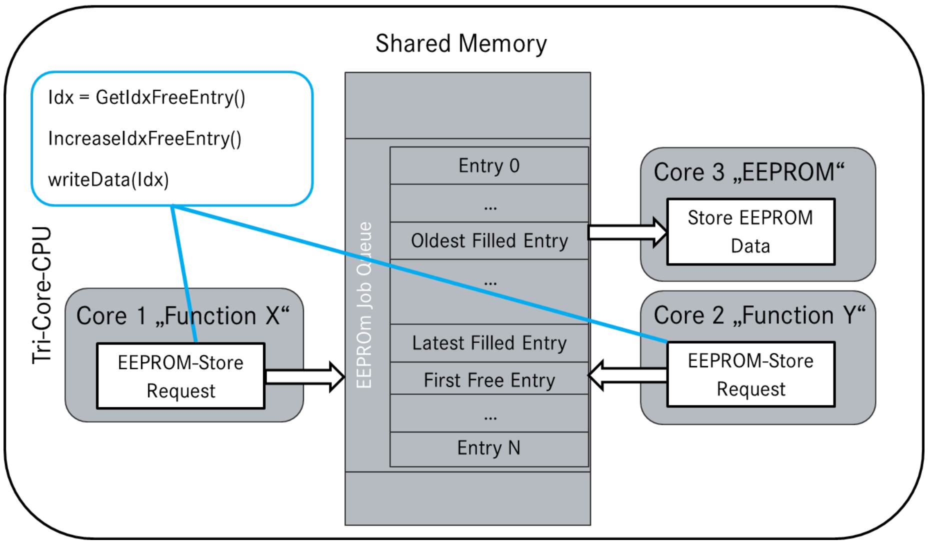


Figure 3: Lost update

In the example shown in Figure 3, the update of the EEPROM is handled in a separate thread (e.g. background/idle thread). If `Function X` and `Function Y` access the queue at the same time (both functions call `GetIdxFreeEntry()` before `IncreaseIdxFreeEntry()` is called), they will use the same Queue index. Result: One function will overwrite the EEPROM request of the other.

Regarding interoperability needs, it is expected that corresponding analysis tools need further specifications in order to detect high level race conditions like exemplified above without incurring too many false-positives. For example, a partial update of data, which is then read by a receiving thread, does not necessarily imply the need for consistency of those data values. In fact this depends on the functional application semantics. So a specification about data dependencies, e. g. list of variables/data to be treated as a unit, is required. Similar considerations apply to non-repeatable reads: It could even be a desired behavior that a thread always gets the most recent update of data whenever reading it during an execution cycle. A specification is required expressing stability needs for data read by a thread in order to decide when a non-repeatable read is indeed an issue.

2.7 Robert Bosch GmbH

To enable automatic verification using static code analysis and formal methods the tools need to be integrated into existing workflows. For obtaining the best possible results it is good to combine different tools having different strengths.

To this end, first of all the static analysis tools' outputs (e.g. reported potential errors) have to be comparable, e.g. the reported error categories have to be standardized. Secondly, also the input to the tools should be defined only at a central location once, in order to be able to meaningfully compare the verification results. Typical inputs are for example environment configurations and used platform.

In general, the overall verification workflow has to be highly automated and hence should require only little manual effort. An important requirement is that it should to be able to run the analysis tools on the same input with none or only minimal effort to fulfill input requirements of the different tools. Moreover, it is required that no changes of the analyzed software itself have to be made. This is to ensure that no errors are introduced or hidden during the analysis. Furthermore, the tools should support the use and import of certain pre-existing models for some parts of the software (e.g. configuration of AUTOSAR OS-behavior).
Finally, for us as Tier1 in automotive it is necessary that all employed tools conform to established automotive safety standards (e.g. ISO26262).

3. Technology provider capabilities

This chapter collects capabilities from technology providers regarding offered methods supporting particular design step(s) in a development process. This includes all kinds of analysis of properties of a design artifact, like absence of deadlocks, fulfillment of a requirement, proving refinement of requirements, synthesis steps computing new design artifacts and/or parameterization of existing ones based on a given set of requirements. The capabilities are described from a methodological perspective and possibly also a technical perspective. That means a technical perspective further refines the described capability by adding technical realization details.

3.1 OFFIS Consistency Analysis

The OFFIS consistency analysis [2] uses bounded model checking technics to find inconsistencies in pattern-based semi-formal requirements. Formal requirements are requirements that are written down in a formal language and thereby allow formal reasoning. A set of requirements is called consistent if there exists at least one run of the system satisfying all requirements.

Our tool takes as an input a set of formal requirements in a pattern language. In a pattern language, patterns are used to formulate requirements. A pattern is a template for a formal requirement and the engineer fills in the parameters of the template with logical formulae over macros. For formal requirements, macros are mapped to inputs and outputs of the system, for semi-formal requirements there is no such mapping. Our tool does currently not support macro definitions so the type of a macro is – depending on the tool configuration – guessed from the context or only Boolean macros are assumed. We support the BTC pattern language [3] version 3.6 and part of the pattern based RSL [4] developed in the CESAR project.

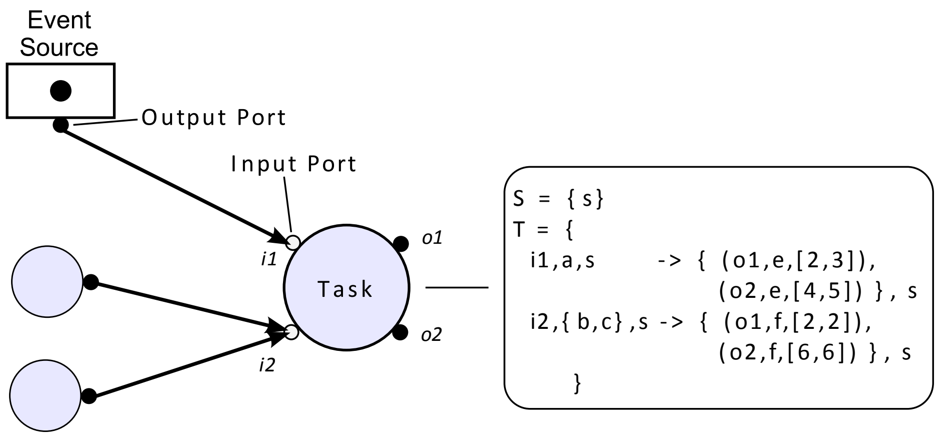
Based on the analysis type, the consistency analysis outputs a maximum consistent or minimal inconsistent sets of requirements which are sub-sets of the input set. Besides, it outputs macro traces that prove consistency of consistent sub-sets.

The OFFIS consistency analysis tools are configured via a graphical interface. We have a simple editor for editing pattern based requirements in that our analysis tool is integrated. The editor uses an RDF [5] based file format. Alternatively, requirements can be imported via OSLC[1] from DOORS NG[2].
The tool provides traces as graphical output that can be browsed in an integrated viewer and exported as HTML.
Besides that, we have a prototypical headless version of the analysis tool that integrates via OSLC in RQM[3] and runs the analysis as an automated test. The configuration is provided via input parameters of the test case and the set of input requirements is stored in DOORS NG and linked to the test case. The results are provided in form of a verdict and detailed information (including graphical representations of traces) as HTML content in the test result.

[1] http://open-services.net/
[2] https://jazz.net/products/rational-doors-next-generation/
[3] https://jazz.net/products/rational-quality-manager/

3.2 OFFIS System-Level Timing Analysis

The system-level timing analysis provided by OFFIS [6] uses model checking techniques to determine typical performance characteristics like response time intervals of tasks and latency intervals of end-to-end effect chains. The analysis requires a *real-time model* as input consisting of tasks allocated on a distributed architecture of ECUs connects via buses. Each task may have a set of input and output ports. Via connections of output ports to input ports, a precedence relation on the tasks is defined. Each input port of a task represents a possible source of activation of a task, meaning it is activated whenever an event is observed on any of its input ports. Input synchronizations are expressed by multiple connections to the same input port. In the example depicted in Figure 4, the task is either activated by an event a observed on the input port i1 or when an event b and an event c are observed on input port i2.



```
S = {s}
T = {
  i1,a,s      -> { (o1,e,[2,3]),
                   (o2,e,[4,5]) }, s
  i2,{b,c},s -> { (o1,f,[2,2]),
                   (o2,f,[6,6]) }, s
     }
```

Figure 4: Task network example

Each activation causes a delay for processing, depending on the activating event and the state of a task. The delays are taken from intervals with best- and worst-case bounds for each output port on which the task sends an event. Referring to Figure 4, when activated by an event a on port i1, the task sends an event e on port o1 after 2 to 3 time units. A characterization of the delay intervals can be obtained from measurement (e.g. by tracing the actual implementation), by analysis, the so-called worst-case execution-time (WCET) analysis, or by estimations (esp. in early phases of development). Where a preceding task is unknown, assumptions about the timing behaviour of the environment can be expressed by means of event sources. Such an event source has parameters like period, jitter, and offset, characterizing an event stream. To define a real-time model, a task network is deployed on a set of resources. Typically, a real-time model contains more tasks than resources, meaning access to resources needs to be scheduled. The scheduling strategy is defined per resource (an ECU or a bus).
Currently, the *real-time model* that shall be analysed can be created via a graphical interface. However the analysis is also available as a library such that it can be also be integrated into other frameworks, allowing a transformation of models like AUTOSAR or AMALTHEA to the input expected by the analysis.

3.2.1. Interface to concurrency defect analyses

Besides typical timing properties like response times per task and end-to-end latencies of functions, the tool can also determine, as a by-product, possible pre-emption scenarios between tasks. Depending on the scheduling strategy it might be the case that a task gets pre-empted by the new activation of a higher priority task. If so, the analysis marks the task executing in the

current state as being pre-empted by the task executing in the next state. This marking is then extended to all other tasks that are active in the current state. So if a task B is executed during the current state and task C is active but not executing and a task A is executed during the next state pre-empting task B, then both tasks B and C are marked as being pre-empted by task A.
Further, the analysis also takes corner cases into account like seemingly simultaneous task activations resulting from the underlying discrete time model. If a set of different tasks are activated at the same discrete step, the analysis assumes that these activations can happen in arbitrary order. This ensures that pre-emption scenarios are sound independently from the length of discrete time slots.
The result of the analysis is a function $PREEMPT: T \times T \mapsto \{true, false\}$ on the set T of tasks allocated to a resource. For each resource such a function is computed. The function assigns to each pair (τ_X, τ_Y) of tasks a boolean value denoting whether a scenario is possible where τ_X can be pre-empted by τ_Y. This function provides valuable insights where concurrency defects might occur and in particular where concurrency defects can be excluded based on the function $PREEMPT$ inferred from the timing behaviour of the system.

3.3 SWEET – SWEdish Execution Time tool

SWEET [7] from Mälardalen University is a tool for static WCET analysis. It can perform two kinds of analyses:

- a detailed program flow analysis, which can find upper and lower loop iteration bounds as well as more complex infeasible path constraints (so-called "Flow Facts"), and
- a simple, not necessarily safe WCET estimation, using simple timing models, suitable for early source-level WCET estimation.

The main analysis method of SWEET is called abstract execution, which is a form of abstract interpretation. SWEET can also perform a number of supporting static analyses, such as a use-def data flow analysis, a conventional value analysis, and program slicing.

SWEET analyses the "ALF" intermediate format [8]. Other formats can be analysed by translation into ALF. A translator from C to ALF exists, as well as experimental translators from the PowerPC and NECV850 binary formats. SWEET also provides a format for "input annotations", which can be used to mark variables as volatile or set input range restrictions on inputs to the analysed program.

SWEET has an expressive native format for Flow Facts, including detailed context information for context-sensitive Flow Facts. SWEET can also export Flow Facts to the AIS format for the WCET analysis tool aiT from AbsInt, and to the flow fact format for the Rapitime tool from Rapita Systems. In this way, the program flow analysis capabilities of SWEET can be utilised also when computing tight and safe WCET bounds by a tool like aiT.

SWEET offers a variety of analysis settings to direct the analysis, allowing to tune for speed and precision. Most of these settings are controlled by command-line flags. SWEET can also read so-called "output annotations" that direct what analysis information that is to be reported, and in which format.

SWEET is open source. Instructions for how to obtain access to the source code can be found at the SWEET web site[4].

3.4 AbsInt – Static Program Analysis

AbsInt is a German SME that provides tools for the validation, verification and certification of safety-critical software. AbsInt's product range includes tools for static program analysis with the goal to obtain sound and precise information on worst-case execution times, maximum stack usage, and potential runtime errors.
The following subsections describe AbsInt's timing analyzer aiT and error analyzer Astrée.

3.4.1. aiT

aiT determines safe and precise upper bounds for the worst-case execution times (WCETs) of tasks in real-time systems [9]. Here, a task means a sequentially executed piece of code (no threads, no parallelism, no waiting for external events, and assuming no interference from the outside). aiT operates on binary executables for selected target architectures. It employs a static program analysis that performs an abstract interpretation [10] without actually executing the program. aiT's results are therefore valid for all possible program runs with all possible inputs.
Main input is a statically linked binary executable containing the task(s) to be analyzed. Secondary input is given by annotations that provide additional information about the analyzed program, e.g. targets of computed calls, loop bounds, and restrictions on the range of variables. aiT tries to compute such information by itself, but sometimes is not able to obtain sufficiently precise useful results.
Annotations may be given in separate annotation files or as specific comments in the C source code. As aiT analyzes binary executables, the presence of C source code is not required, but if it is available, it is read by aiT to watch out for embedded annotations and to be able to refer to C source code in its output.
aiT also requires information about the hardware configuration. Such information can be specified by options in the graphical user interface (GUI), textual descriptions in an annotation file, or specification of the contents of configuration registers (details depend on the target architecture). Names of executable and annotation files are entered in the GUI. From this information, a project file can be formed.
aiT is part of the AbsInt a^3 analyzer framework, which comes with a graphical user interface (GUI). Thus, aiT can be started for interactive work by starting the a^3 GUI, loading a project file, and starting a WCET analysis. The a^3 tool can also be started in batch mode without user interaction under control of a project file.
aiT produces safe over-approximations of the overall worst-case execution time (WCET), the WCETs for routines and basic blocks, worst-case execution numbers for routines and basic blocks, and the worst-case path. This information is given in a textual report file for human inspection and an XML report file that may be read by other applications. The a^3 GUI also offers various tables and charts with analysis results, and combined call graphs and control-flow graphs showing the structure of the analyzed program with analysis results attached to the structure elements.

[4] http://www.mrtc.mdh.se/projects/wcet/sweet/

3.4.2. Astrée

Astrée is a static program analyzer that has been developed by ENS and licensed by AbsInt for industrialization and also addition of new features in cooperation with ENS and UPMC. Astrée finds runtime errors and invalid concurrent behaviour in safety-critical embedded applications written or generated in C. This is done by static program analysis by means of abstract interpretation. The analysis covers all possible program runs with all possible inputs without actually executing the program. Astrée is sound: if no errors of a certain class are signalled, the absence of errors from this class has been proved. Astrée also includes a rule checker for checking coding rules such as the MISRA rules.
Astrée is used in WP2 for finding runtime errors in sequential code and in WP5 for finding potential data races, lock/unlock problems, and further invalid calls to OS services.
Astrée offers an interactive mode with a graphical user interface for setting analysis parameters that can be stored in a project file, and a batch mode that works without user interaction under control of a project file. The main input of Astrée is C source code, either original or already preprocessed. Secondary input is given by directives that provide additional information about the analyzed program. Directives may be inserted into the C source code or given by special files containing annotations. An annotation consists of a directive and a description of the program point to which the annotation applies. Directives can be provided by users, but Astrée also offers interfaces to certain model-based code generators. These interfaces automatically convert relevant model information into Astrée annotations (see also Section 4.2.2).
To better support the analysis of applications running under the OSEK operating system, Astrée was extended by an Open Image Library (OIL) converter that extracts all information specified in an .oil configuration file and automatically generates the corresponding C data structures and access functions.
Astrée produces a textual report file for human inspection and an XML report file that may be read by other applications. Various custom reports can be generated that contain specific information of interest.

3.5 MES Quality Commander for online quality monitoring

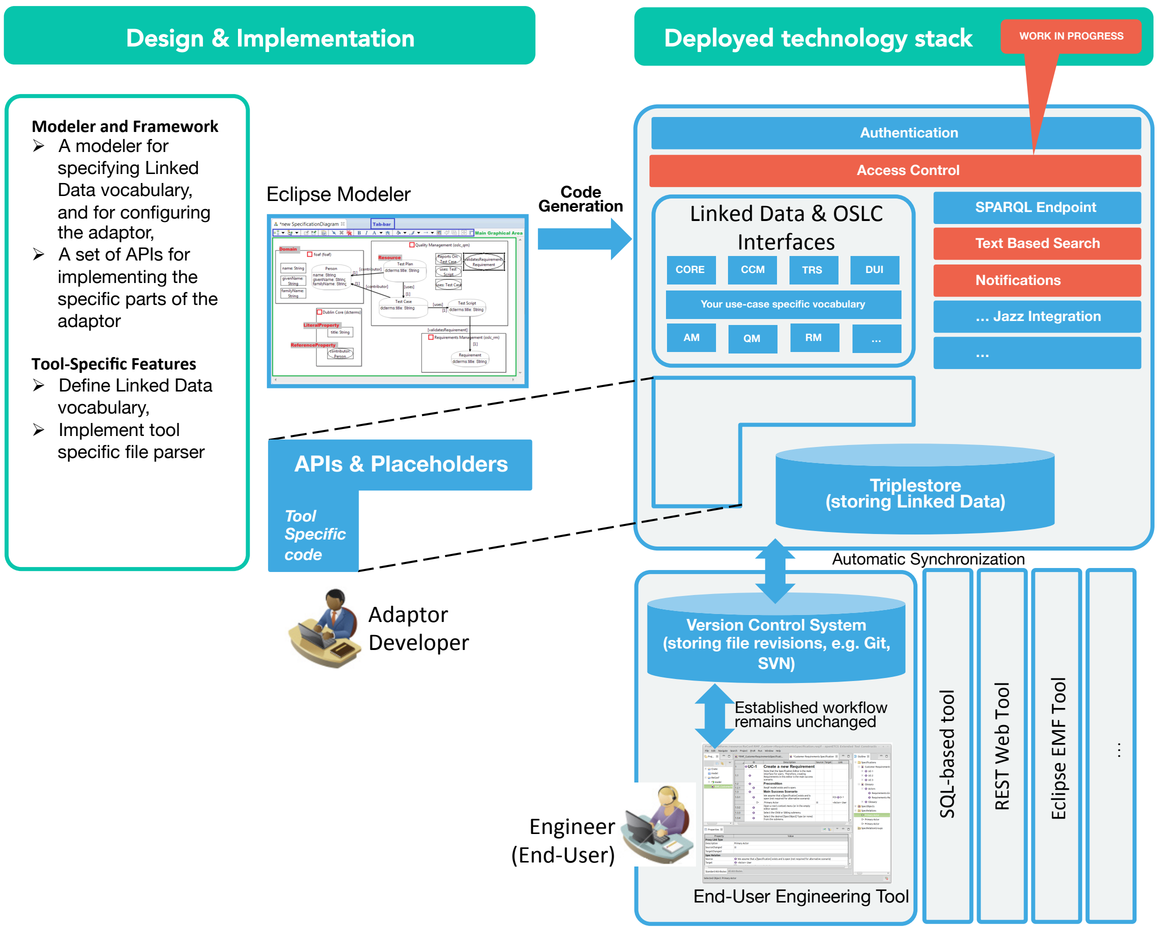
MES Quality Commander® (MQC) is a dynamic quality monitoring and management tool for software development that captures all the decision-making data that you need throughout the software lifecycle. MQC computes and evaluates the quality and product viability of software on the basis of all relevant development artifacts and the corresponding key performance indicators (e.g. guidelines, complexity, tests, coverage and reviews). Comprehensive quality assessment of software development in the sense of well-known norms such as ISO 26262 and ASPICE, respectively, is provided. User-friendly visualizations of product maturity, weaknesses and need for action during any stage in a project increase the development and product value of safety relevant software.
MQC does also optimize return on investment by perpetual availability of trend analysis that indicates product maturity and achievable level of product quality. An efficient comparison of quality and progress for different development projects ensures error proofing very early. Project-specific evaluation with individually configurable quality models adaptable to company-specific business processes and in compliance with ISO and ASPICE simplify quality assurance of safety relevant software development. Various possibilities of exporting (e.g. Power Point, PDF, HTML) facilitate reporting and thus effort and charges can be controlled and minimized by MQC. The Web Viewer guarantees worldwide access to your project. By batch automation MQC is able to

collect all project relevant tool reports nightly so that currency of quality dashboards is provided out of the box.
In order to assist MQC it is recommended but not necessary to provide tool reports in an MQC XML format. Of course, other formats like Excel, Text and several Databases are supported.

3.6 KTH – OSLC / Linked Data Adaptors Development Kit

KTH supports the use of the OSLC[5] and Linked Data[6] open standards for lifecycle interoperability across engineering tools and data repositories used within a development environment for systems engineering (or tool-chain). Thanks to these standards, engineering data coming for multiple and heterogeneous sources can be exposed, queried and modified according to standardized integration APIs based on HTTP. At KTH, we are developing development kits aiming at easing the development of OSLC and Linked Data *adaptors* on top of these heterogeneous data sources. An illustration of this development kit is given on the figure below. The first step of the process consists in defining the Linked Data vocabulary that will be exposed by the adaptor, provided as an Eclipse Modeler[7]. Using the latter, one can define arbitrary complex linked data structures according to the RDF principles[8]. The tool embeds a code generator used for generating automatically a Java skeleton implementing the basic capabilities of the adaptor's front-end to be deployed (see right-hand side of the picture, e.g., with serialization / deserialization of the RDF resources in various data formats like JSON or RDF/XML, basic query mechanisms, basic OSLC delegated web-based user interfaces). The development kit is optionally packaged with a triple store (i.e., a purpose-built RDF database for the storage and retrieval of triples through semantic queries) that can be used to cache RDF data. Depending on the technologies used by the underlying tools or repositories, the interactions with the latter and the adaptor can be done in different ways. For instance, if the end-users work with workspace-based tools coupled with version management systems (e.g., MATLAB with an SVN repository, or Eclipse with a Git repository), we provide back-end plugins for our development kit easing the integration process of the adaptor with these systems. Other back-end plugins are also being implemented, e.g., for SQL databases or EMF-based tools.

[5] http://open-services.net
[6] http://linkeddata.org
[7] https://wiki.eclipse.org/Lyo/ToolchainModellingAndCodeGenerationWorkshop
[8] https://www.w3.org/RDF/



Design & Implementation
Deployed technology stack
WORK IN PROGRESS
Modeler and Framework
A modeler for specifying Linked Data vocabulary, and for configuring the adaptor,
A set of APIs for implementing the specific parts of the adaptor
Tool-Specific Features
Define Linked Data vocabulary,
Implement tool specific file parser
Eclipse Modeler
Code Generation
APIs & Placeholders
Tool Specific code
Adaptor Developer
Authentication
Access Control
Linked Data & OSLC Interfaces
CORE
CCM
TRS
DUI
Your use-case specific vocabulary
AM
QM
RM
...
SPARQL Endpoint
Text Based Search
Notifications
... Jazz Integration
...
Triplestore (storing Linked Data)
Automatic Synchronization
Version Control System (storing file revisions, e.g. Git, SVN)
Established workflow remains unchanged
Engineer (End-User)
End-User Engineering Tool
SQL-based tool
REST Web Tool
Eclipse EMF Tool
...

Figure 5: KTH OSLC / Linked Data Adaptors Development Kit

3.7 KIT and FZI: Traceability for Static Code Analysis

The static code analysis tool from KIT called “Low-Level Bounded Model Checker” (LLBMC) analyzes C(++) source code files and finds: arithmetic errors, bit-operations-errors, memory errors and customer defined assumptions. Therefore, a target architecture can be defined via configuration settings. This configuration is the system context for the software. As a static analysis, the software isn’t executed during the analyzation.

3.8 FindOut Technologies AB – visualization of linked data resources

FindOut Technologies AB is an SME providing product development organizations with methodology and technology to improve efficiency. We support the use of Linked Data and OSLC for interoperability across the engineering tool-chain, and have experience from developing OSLC adaptors for various engineering tools. Within the Scania use case, FindOut is providing technologies for mapping data resources to visual objects in order to obtain flexible visualizations of a wide variety of data. Our visualization technology provide traceability and a dynamic visual representation of interconnections, hierarchies and dependencies of engineering tool’s entities.

FindOut visualization solutions are based on proprietary 3rd party graphical frameworks as well as in-house developed Open Source javascript frameworks that can be reused for other engineering use cases.

3.9 B&M Code Security Analysis

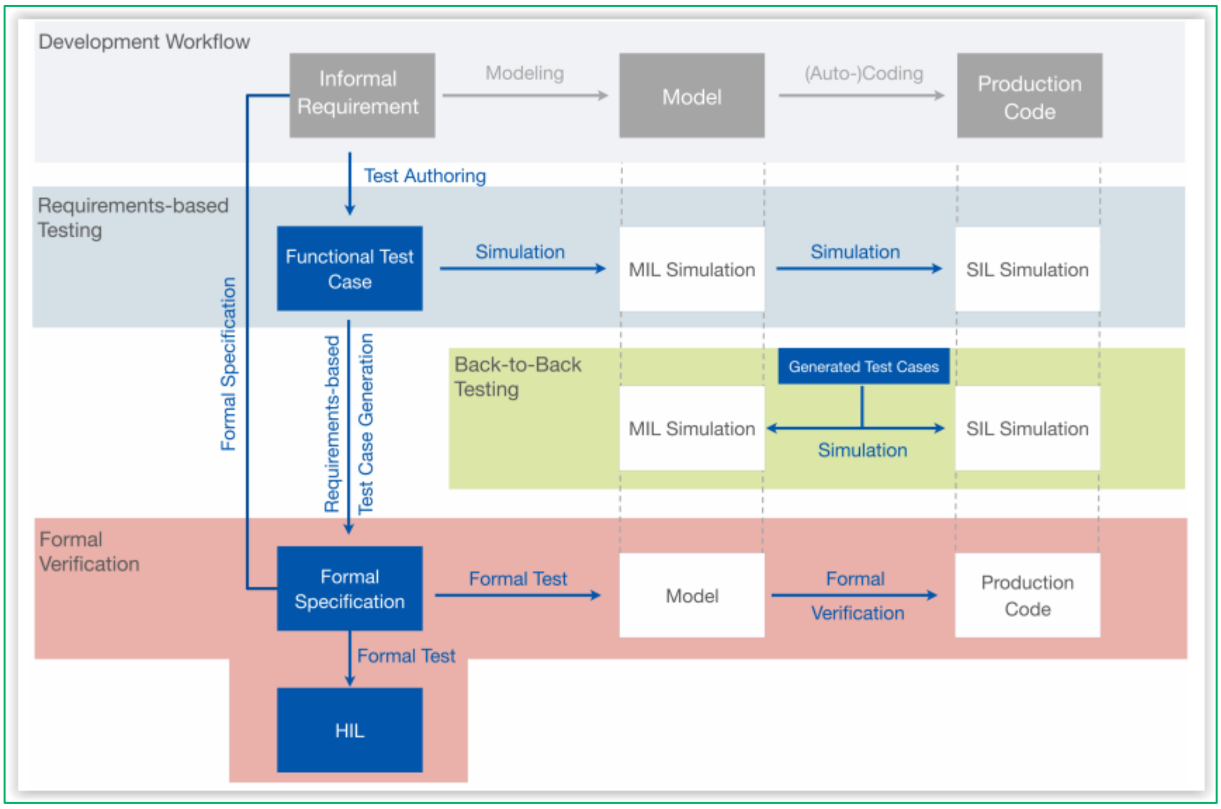
The B&M Code Security Analysis uses a B&M-defined, guided assessment process to identify potential security issues and attack scenarios from a set of general implementation defects provided by general static software analysis tools. The analysis is based on precise review criteria for selected results of these analysis tools to classify them as true-positive security weaknesses.

The analysis takes as input the system's implementation and a so-called system build specification. A system build specification includes (among others) a definition of the system borders, a definition of the relevant system variant and external libraries used in the project. As a result, the method provides a set of true-positive security vulnerabilities which may actually enable successful attack scenarios.

The method is semi-automatic. As a first step, a set of general potential defects and weaknesses are identified in the codebase. This detection is done using general static software analysis tools. In a second step, a selected subset of these detected issues are manually reviewed and classified according to well-defined security review guidelines.

3.10 BTC-ES: MBT and Formal methods support

BTC EmbeddedPlatform addresses different processes and methods used for the verification of reactive embedded systems modeled with Simulink / TargetLink and/or C-code. The addressed use-case span over several stages of a V-model based development and testing cycle.



- Requirement-based Testing for software unit-testing and integration testing including traceability between created test cases and informal requirements captured in IBM DOORS, PTC Integrity, dSPACE SYNECT or Microsoft Excel, structural coverage computation for models and code.
- Semi-formal and formal Specification of functional requirements for unique requirement understanding and reuse in other process steps to increase quality and to further automate adjacent verification processes.
- Formal Testing as addition to traditional requirement-based testing to take advantage of formalized requirements for parallel investigation of all requirements and automatic verdict computation including detailed measurable requirements coverage, applicable as co-simulation during MIL, SIL, PIL and HIL Testing with established dSPACE HIL solutions.
- Automatic Test Case Generation for functional requirements out of formal requirements including different degrees of requirement coverage.
- Mathematically complete Formal Verification of functional requirements using model checking technology for Production C-Code and TargetLink models.
- Back-to-back testing for checking equivalence between Simulink models, TargetLink models and C-code for protecting model-based development of ECU functions or migration to different MATLAB / TargetLink versions.

An essential aspect is the integration and interconnection of the capabilities within such a platform and from outside with the platform. It is getting more and more crucial for successful and efficient product engineering. Hence, we design the platform in a way that all use cases share the same data model, allowing a very tight integration and smooth workflow. Additionally the platform is designed to be very open such that different external tools like DOORS or PTC Integrity can be integrated with various integration paradigms, for instance import/export, or OSLC based, etc. Other analysis tools can also designed as add-ons on the platform to provide further smoothly integrated capabilities.

4. Benefits of enhanced interoperability

This chapter collects needs on interoperability from technology providers regarding offered methods supporting particular design step(s) in a development process. This includes all kinds of analysis of properties of a design artifact, like absence of deadlocks, fulfillment of a requirement, proving refinement of requirements, synthesis steps computing new design artifacts and/or parameterization of existing ones based on a given set of requirements. Based on the description of the capabilities of a method in Chapter 3, needs on interoperability are described, which would lead to an improvement with regard to 1) the applicability of the method 2) reducing pessimism of the results of an analysis method.

4.1 OFFIS Consistency Analysis

Knowledge about value ranges for macros can gain performance of the tool. For partial consistency – which is an extended form of consistency that also takes simultaneous occurrence of events into account – additional knowledge about the occurrence of external events can reduce false warnings. Both are information that may be available from dedicated analysis of the interface specification of components.

4.2 AbsInt – Static Program Analysis

The AbsInt tools can profit from interoperability in several ways.

4.2.1. Combined System and Code Level Timing Analysis

A code-level timing analyzer such as AbsInt's aiT computes a safe upper bound for the worst-case execution time (WCET) of a task, assuming no interference from the outside. Effects of interrupts, IO and timer (co-)processors are not reflected in the predicted runtime and have to be considered separately within a system-level timing analysis, which computes the worst-case response times (WCRTs) of an entire system from the task WCETs and information about possible interrupts and their priorities.
So code-level timing analysis and system-level timing analysis complement each other nicely. This is the reason why an interface between aiT and the system-level tool SymTA/S by Symtavision was established in the INTEREST project and improved in the INTERESTED and ALL-TIMES projects [11]. This interface is an open data exchange mechanism and format called XTC, which can be (and has been) adopted by other tools (see Section 5.2.1). With XTC, users of SymTA/S can cause aiT to perform timing analyses at the code level, whose results are mapped back to the scheduling tool in a fully automatic way. Thus overall timing analyses can be performed.

4.2.2. Static Analysis in Model-Based Software Development

Often, software to be analyzed with aiT or Astrée is not written by hand, but generated from a model by means of an automatic code generator such as SCADE from Esterel or TargetLink from dSpace. In such a case, users of the static analysis tools can profit from an integration between the analyzers and the modelling tools. The modelling tools can trigger the analysis of pieces of code of interest and pass information to the analyzer for improving its performance and the precision of its results. The analysis results can be communicated back to the model level so that they are presented directly in the user interface of the modelling tool. The resulting combination

allows for the development of more secure and better-performing systems while decreasing time-to-market through enhancing development productivity.
Within the INTEREST project, aiT has been integrated with ASCET, a model-based design tool with automatic production code generation [12], and the SCADE Suite, a model-based design tool with an automatic code generator SCADE KCG qualified as a development tool w.r.t. DO-178B level A [13]. ASCET is provided by the company ETAS and is widely used in the automotive domain. The SCADE Suite is provided by Esterel and is widely used in the avionic domain.
The integrations ASCET/aiT and SCADE/aiT have been designed in a way that the analysis results for code generated by ASCET and SCADE are conveniently accessible from within the respective graphical user interfaces. The aiT analyses run completely automatic without any user interference. Results of practical experiments show a good precision. For the developer, the immediate and detailed feedback provided by mapping back aiT's results into the IDE of design tools helps to find the critical areas of the project where most of the resources are spent.
Astrée can be used in both model-based and classical development. Directives can be provided by users, but Astrée also offers interfaces to certain model-based code generators. These interfaces automatically convert relevant model information into Astrée annotations.
In the TIMMO-2-USE project, an integration between Astrée and Targetlink from dSPACE was set up [14]. Astrée can be called from within Targetlink to analyze C code generated from the Targetlink model. Astrée has access to Targetlink's data dictionary to obtain information on the generated code. If Astrée is called from Targetlink, it is able to trace its findings back to the model level. Tracing of requirements is not supported.

4.2.3. Static Analysis and Model-Based Testing

A drawback of Astrée's method of static analysis by abstract interpretation is that there can be false alarms caused by the abstraction mechanism, i.e. spurious notifications about potential run-time errors that are not actual bugs. Therefore, all alarms have to be investigated by the developers to determine whether they correspond to true errors which have to be fixed, or whether they are false alarms. This can cause significant effort. A way to reduce this effort is to apply model-based testing to automatically find test cases for alarms reported by Astrée.
Therefore, Astrée has been integrated with BTC EmbeddedTester in the course of the TIMMO-2-USE and MBAT projects [15]. When EmbeddedTester finds a test case reproducing a potential error found by Astrée, it has not only been proven that it is a true error, but users can directly investigate the situation in a debugger. When no test case reproducing the error could be found, the interpretation depends on the test model generation: when full test coverage can be achieved, the absence of the error has been proven.
Situations where no full test coverage was possible, or where the error could not be reproduced in the given amount of time, have to be manually investigated - but even here the test coverage obtained is a valuable feedback for the user. With this coupling the effort for alarm analysis can be significantly reduced. Preliminary experimental results demonstrate the viability of the approach.

4.3 MES Quality Commander for online quality monitoring

The integration of multiple analysis tools provides comprehensive information on the quality of development artefacts. Trends and shortcomings in quality can be identified and corrective measures can be determined. By importing the XML-based output of analysis tools, the MES Quality Commander will automatically assess the overall quality such that the development team is able to determine these corrective actions.

4.4 KIT and FZI: Traceability for Static Code Analysis

The middleware approach to couple different development tools based on OSLC especially in context of static code analysis and results traceability is an extension to existing traceability approaches. Comparison from different analysis tools can be fulfilled within this concept, as well as traceability between different static analysis tools. Additionally, this decentralized middleware approach allows integration of solution-specific methodologies and tools based on a consistent and comprehensive data link. This greatly supports collaboration for world-wide development without loss of traceability.

4.5 Code Security Analysis

Interoperability between the B&M Code Security Analysis and supporting analysis tools provides many advantages over a purely manual application of the method without automated tool support. Besides increased efficiency and effectiveness, results based on automated tools are reproducible, more reliable and comparable among different projects and reviewers. Especially comparability among projects enables different application scenarios for automotive OEMs.

4.6 BTC-ES: MBT and Formal methods support

As discussed in our previous section in chapter 3 our BTC EmbeddedPlatform addresses different processes and methods used for the verification of reactive embedded systems modeled with Simulink / TargetLink and/or C-code. Hence, an essential aspect is the integration and interconnection of the capabilities within such a platform and from outside with the platform. It is getting more and more crucial for successful and efficient product engineering. Hence, we design the platform in a way that all use cases share the same data model, allowing a very tight integration and smooth workflow. Additionally the platform is designed to be very open such that different external tools like DOORS or PTC Integrity can be integrated with various integration paradigms, for instance import/export, or OSLC based, etc. Other analysis tools can also designed as add-ons on the platform to provide further smoothly integrated capabilities. OFFIS is a good example in this project, because they plan to implement analysis capabilities on top of the platform. Also with an integration with MES Quality Commander we can provide additional value to the users.

5. Synthesis of interoperability benefits

Here we identify synergies from tools and use cases. We identify common parameters and reusable data sets among tools and use cases.

From both the users and the tool-providers perspective several requirements on tool interoperability that are at the same time benefits are expected from interoperability of analysis and synthesis tools:

- **Traceability.** (Sections 2.1, 2.4, 2.5 and 2.7) Analysis results shall be traceable back to the system model / the code / the requirements. This is required by various safety standards and helps to ensure validity of data throughout the complete design process. Furthermore, traceability is required for a backward analysis mapping (possible) faults to model elements and code analysis and for a forward analysis determining the impact of possible faults and changes on design artifacts in further steps.
- **Comparability.** (Sections 2.1, 2.2, 2.7) Interoperability between tools allows to compare the output of different tools. On the other hand, the output of different analysis tools needs to be comparable. Comparability helps to profit from different strength and capabilities of various analysis tools.
- **Reproducibility.** By smoothly integrating synthesis and analysis tools, transformation of data between tools is done more automatically and less error prone. Global configuration data shall be reused between runs and tools. This makes the inputs and thereby the outputs of tools reproducible. (Sections 2.2, 4.5)
- **World-wide collaboration.** There is a growing internationalization of development teams (Section 2.1) which requires the possibility to collaborate world-wide.
- **Automatization.** It shall be possible to automate the execution of analysis tools. This reduces the effort compared to manual execution and enables fast refinement of development artefacts. (Sections 2.2, 2.3, 2.7)
- **Standardized Data Formats.** The need for standardized data formats has been identified in Sections 2.1, 2.2, 2.3 and 2.7 as an enabler for automatization and to reduce the effort to setup an analysis.

As a general goal, interoperability significantly reduces the effort to setup and execute an analysis, which is KPI 2.1 in ASSUME. Furthermore, interoperability can improve the precision of an analysis (KPI 2.2) and help to identify false warnings (KPI1.3).

5.1 Interoperability Scenarios

From the descriptions of available tools and scenarios in ASSUME in Sections 3 and 4 we can consolidate the following interoperability scenarios.

- High-level modeling and engineering tools produce executable code and annotations for a timing analysis. The results are traceable back to the model elements and displayed in the modeling tool's IDE (see Section 4.2.2).
- Flow facts from code analysis supply WCET analysis on binary level (Section 3.3).
- WCET analysis results on task level are used to form WCRT results on system level.

- False alarms from static analyses are found via model-based testing with generated test-cases (Section 4.2.3).
- Quality assessment tools collect analysis results and display them in a user-friendly, condensed manner to produce trends and project status (Section 3.5).

To profit from the tool interconnections above, the tools need to use common formats for data exchange as well as the data flow and tool execution needs to be coordinated. Here exist different approaches:

- Tools share a common platform; either is this considered during tool development as in the BTC tools or a plugin architecture (Section 3.10).
- Analysis tools run headless, and configuration and results are integrated into a modeling tool's IDE (see Section 4.2.2).
- An interoperability standard, i.e. OSLC, is used. For this, tools must either provide an interface to the standard or adapters wrap the tools to provide the interface. This approach is explained in detail in Section 5.1.1.

5.1.1. KIT and FZI: Traceability for Static Code Analysis

The FZI works in close cooperation with QPR and KTH in the context of WP3 on a solution to interchange the analysis results and the system configuration via a middleware framework such as Open Services for Lifecycle Collaboration (OSLC). Therefore, core concepts of the interoperability format form the basis for services and exchange with other development artifacts in order to achieve traceability for results between different tools.
Figure 6 and Figure 7 show two possible application scenarios. There are clients (green boxes) and server (blue boxes). The arrows describe RESTful communication between server and clients. The OSLC adaptor (red boxes) are for both the clients and the server and supports the interoperability with the OSLC specific service concept. The OSLC service concept uses RESTful communication and the resource description framework (RDF).
In both application scenarios, there are three types of frameworks and tools used in development of automotive systems: 1) IDE tools for implementation of software such as Eclipse or Microsoft Visual Studio, 2) static code analysis tools such as LLBMC from KIT or QPR REFINE from QPR and 3) design and architecture tools such as Enterprise Architect from Sparx Systems or PREEvision from Vector Informatik. In further steps, requirements engineering tools such as IBM DOORS or again Enterprise Architect from Sparx System will also be integrated into the approach. The use of requirement tools could lead to additional traceability capabilities in this application scenario, which allows change impact analysis. These tools define the data on the client side. According to the clients, there are servers, which keep and store the data from the clients.
The figures with the application scenarios shows two adaptor concepts. The first concept is that the adaptor runs as an external program, which communicates over an interface on the client side with the client. The static code analysis client shows this as an example in Figure 6 and Figure 7. Therefore, it is possible in the early concept phase to use e.g. the analysis results as XML files as data source from the analysis tool and send it to the analysis server. The second concept is to integrate the adaptor directly into the client tool, as shown in the figures at the design and architecture tool. This approach works without further interfaces, but needs a close integration into the tool from the OSLC adaptor during the tool development in order to be able to execute the analysis tool from external tools.
The green arrows shows the client server communication between the tool client and the relative tool server. The blue arrows shows the requests between the different tools. The static code analysis tool requests for files from the file server and information about the software and

hardware configurations from the design and architecture server. The design and architecture tool requests for implementation of the software architecture.

In application scenario AS1, the static code analysis tool works on the client side. In application scenario AS2, the static code analysis tool works directly on the server. The advantage in application scenario AS2 is that the analysis tool can get an analysis job directly from the IDE or a design tool or in a further application scenario form a build and test server.

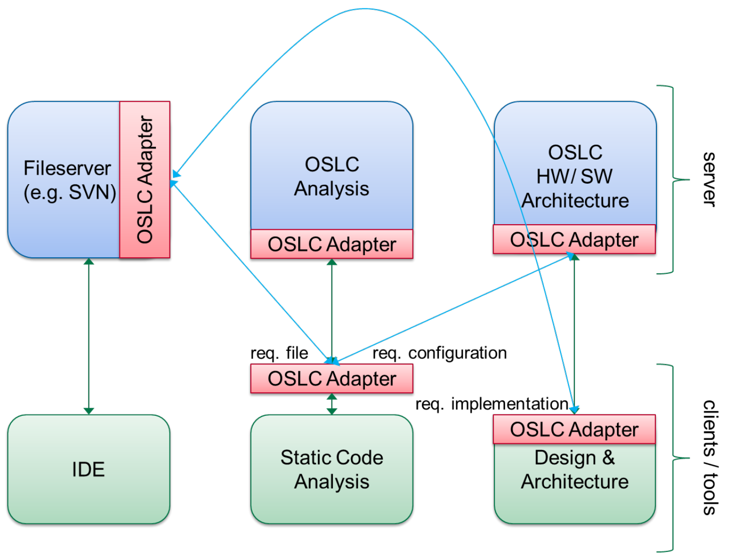


Figure 6: AS1 - Interoperability concept with an IDE, a static code analysis tool and a design and architecture tool on the client side

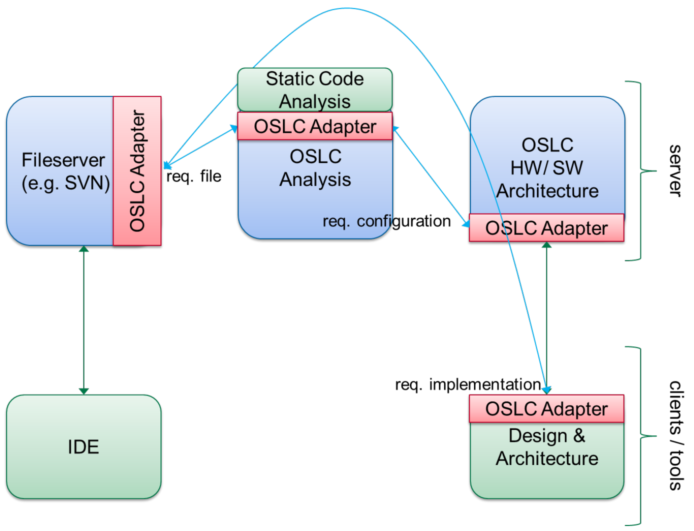


Figure 7: AS2 - Interoperability concept with an IDE and a design and architecture tool on the client side and a static code analysis tool on the server side

For prototypical implementation of the application scenario, established tool suites and frameworks for OSLC and the in the application scenario described type of tools are used. Within the project, KTH is refining the development tool for model-based description of OSLC resources and communication, which serves as a basis for close cooperation in conceptual description of the application scenario. The KIT defines the interchange formats for static code analysis tools, which serves as a basis for the resources. The fileserver and the analysis tool as described in the analysis scenario AS1 will be integrated in a first step from the FZI. In following steps the FZI investigate the tool integration with a design tool and a requirement analysis tool.

5.2 Exchange Formats

A basic enabler for tool interoperability are common data formats. In an automated environment (e.g. OSLC) common formats for configuration and results are necessary for the data flow.

From the end users' needs (Section 2) we can infer that source code is present in C or C++. However, as observed by Berner & Mattner in Section 2.2, the used language versions and compilers differ. C-code is either generated from Simulink or SCADE models or written by hand. Timing analysis tools (e.g. WCET) operate on the C-code or binaries.

While the input formats highly depend on the use-case (see Section 2), a need for tool independent formats for configuration and analysis results has been identified. Timing analysis tools for example require detailed information on the target environment as well as additional source code annotations. Although both SWEET (Section 3.3) and AbsInt tools (Section 3.4) support annotations directly in the C-code as well as separate annotation files, the concrete formats differ. To enable data exchange between tools different strategies exist.

- Tools are integrated into one platform and thereby sharing one data model. This is either the case for tools developed by one vendor, e.g. the BTC tools (Section 3.10) or different tools are integrated into one IDE as done in the INTEREST project (Section 4.2.2).
- Tool vendors adapt each other's formats. For example, SWEET supports flow-fact formats from aiT and Rapitime (Section 3.3). The MES Quality Commander and BTC EmbeddedPlatform support a wide range of different input formats to fit for many users' needs. OFFIS supports the BTC pattern library to make the Consistency Analysis applicable to requirements formalized with BTC EmbeddedSpecifier.
- Tool vendors agree on a common exchange format. This has been done with the XTC format (Section 5.2.1) between aiT and SymTA/S (Section 4.2.1). The ASSUME SCA (static code analysis) tool exchange format (Section 5.2.2) for configuration and analysis results that will be used in the FZI/KIT scenario will be defined from QPR in WP2, with consideration of common properties and capabilities of static code analysis tools.

5.2.1. XML Timing Cookies (XTC)

The communication between the analyzers aiT and/or Astrée and the other tools described in Section 4.2.1) is based on a standardized exchange format, called XTC [16]. The XML Timing Cookie (XTC) language is an XML application conforming to the Extensible Markup Language 1.0 (Fourth Edition). XTC was originally developed with a focus on timing tools in the INTEREST project, but was then enhanced to cover analysis and verification tools in general in INTERESTED, ALL-TIMES, TIMMO-2-USE, and MBAT. It is a generic interchange format to transport information, analysis requests, and results between the involved applications.
The design of XTC is based on two observations: First, the information flow between tools often is cyclic, suggesting a request-response mechanism. For example, a scheduling tool or code

generator may issue a sequence of requests to compute the WCET of the relevant tasks or subsystems. Second, each tool requires a potentially large set of data about the system under design where intersections between different sets of data are small.
An important goal of XTC is to avoid duplicating sophisticated interface specifications for each specific tool. The concept of XTC therefore allows developers to specify tool-specific information once with the corresponding tool-specific mechanisms and to store this information for the next communication rounds between the tools. This is similar to repeatedly visiting a web site that requires certain user information. Such information is typically stored in a cookie and retrieved when the user visits the site again.
To match these requirements, an XTC document is made up of two parts: a common section and a cookie section. The common section contains information about a specific analysis request from one tool to another tool, and additionally holds the response to that request when the result is returned. One cookie section per communicating tool holds tool-specific information required to service the request. As an example, the cookie section for aiT typically holds details about the hardware configuration, and the detailed analysis option settings. While all tools have access to the common section, the cookie sections are private to the tools concerned. This way, key information is stored in one place (the common section), while tool-specific information can be refined iteratively and is only modified by the corresponding tool.
To invoke an aiT analysis from TargetLink, SCADE, or SymTA/S, an XTC file is created, containing items such as the name of the executable to be analyzed, the selected entry point (runnable), the appropriate machine settings file, and possibly additional information useful for aiT. This XTC file is also reused for subsequent invocations in order to preserve any parameters that have been selected manually during the first tool run. For an Astrée analysis from TargetLink, the XTC file contains the list of source files to be analyzed, the location of the XML export of the Data Dictionary, include directories and defines necessary for preprocessing, external Astrée directives, etc. The tool parameterization has to be done only once; all further invocations can be done fully automatically in batch mode without any tool-specific user interaction.

5.2.2. Comparison to ASSUME SCA tool exchange formats

The ASSUME SCA tool exchange formats are still under development but the purpose and structure is different to XTC. In contrast to XTC the ASSUME format is divided into two specification, the Configuration format and the Reports format.
In general, the XTC format focuses on timing analysis and related tasks such as determining loop bounds. The result types defined in XTC are special to these tasks and tool independent. In the ASSUME SCA format, the structure for the analysis results is optimized for providing traceability from defects to precise locations in the source code instead of exchanging results between tools. To issue a request for a supporting analysis as in XTC is not suggested. Also, the tool independent configuration is more fine-grained though tool-specific configuration is supported. Furthermore, using the XTC format tools may be started using interactive mode to complete the configuration. The ASSUME SCA formats however require that a tool is fully configured in the configuration file which is necessary to be run headless within the OSLC framework.

6. Conclusions and Discussion

This deliverable brings together the end-users' needs, the technology providers' capabilities, expectations, and experiences on interoperability present in ASSUME, mainly between WP3 partners, but also from AbsInt, TNO and Daimler. Section 5 describes at least part of the activities within ASSUME, as well as existing solutions, to provide tool interoperability.

For one of the main goals of interoperability – automated toolchains – OSLC (see Section 5.1.1) seems to be a promising way. However, results from related projects show alternatives that are worth considering.

Common data formats for tool configuration and analysis results are a basic enabler for interoperability—in the OSLC approach as well as its alternatives. Because tools and use-cases are heterogeneous, finding a common format that fits all needs is difficult. Section 5.2.1 describes XTC as an existing inter-tool exchange format for timing analysis. XTC is compared to the planned ASSUME static code analysis tool exchange format which is more focused on traceability.

Because of the different use-cases, needs, and methods within ASSUME, it is not possible to propose one interoperability solution for ASSUME. Because the activities in ASSUME are evolving, the final interoperability solutions need to be presented in a later deliverable. However, this deliverable confirms the need of interoperability and shows possible and current ways to go.

References

[1] *ISO 26262-1:2011-11, Road vehicles - Functional safety - Part 1: Vocabulary,* Beuth Verlag GmbH.

[2] C. Ellen, S. Sieverding and H. Hungar, "Detecting Consistencies and Inconsistencies of Pattern-Based Functional Requirements," in *Formal Methods for Industrial Critical Systems - 19th International Conference, FMICS 2014*, Florence, Italy, 2014.

[3] H. J. Holber en S. Häusler, „From Safety Requirements to Safety Monitors - Automatic Synthesis in Compilance with ISO 26262," in *Embedded World Conference 2012*, 2012.

[4] P. Reinkemeier, I. Stierand, P. Rehkop en S. Henkler, „A pattern-based requirement specification language: Mapping automotive specific timing requirements," in *Software Engineering 2011 - Workshopband (inkl. Doktorandensymposium), Fachtagung des GI-Fachbereichs Softwaretechnik, 21.-25.02.2011, Karlsruhe*, Karlsruhe, 2011.

[5] F. Manola, E. Miller en B. McBride, „RDF 1.1 Primer," 2014.

[6] I. Stierand, P. Reinkemeier, T. Gezgin and P. Bhaduri, "Real-Time Scheduling Interfaces and Contracts for the Design of Distributed Embedded Systems," *8th IEEE International Symposium on Industrial Embedded Systems (SIES'13),* pp. 130-139, 2013.

[7] B. Lisper, „SWEET - A Tool for WCET Flow Analysis (Extended Abstract)," in *Proc. 6th International Symposium on Leveraging Applications of Formal Methods (ISOLA'14)*, Corfu, Crete, 2014.

[8] A. E. B. L. C. S. a. L. K. Jan Gustafsson, „ALF - A Language for WCET Flow Analysis," in *Proc. 9th International Workshop on Worst-Case Execution Time Analysis (WCET'2009)*, Dublin, Ireland, 2009.

[9] C. Ferdinand, R. Heckmann, M. Langenbach, F. Martin, M. Schmidt, H. Theiling, S. Thesing and R. Wilhelm, "Reliable and precise WCET determination for a real-life processor," in *Proceedings of EMSOFT 2001, First Workshop on Embedded Software, LNCS 2211*, 2001.

[10] P. Cousot en R. Cousot, „Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints," in *Proceedings of the 4th ACM Symposium on Principles of Programming Languages*, Los Angeles, California, 1977.

[11] C. Ferdinant, R. Heckmann, M. Jersak, F. Martin en K. Richter, „Integrating system-level and code-level timing analysis for dependable system development," in *4th European Congress ERTS Embedded Real Time Software*, Toulouse, France, 2008.

[12] C. Ferdinant, R. Heckmann, H.-J. Wolff, C. Renz, M. Gupta, O. Parshin en R. Wilhelm, „Towards integrating model-driven development of hard real-time systems with static program analyzers," in *SAE 2007*, Detroit, USA, 2007.

[13] C. Ferdinand, R. Heckmann, T. L. Sergent, D. Lopes, B. Martin, X. Fornari en F. Martin, „Combining a high-level design tool for safety critical systems with a tool for WCET analysis on executables," in *4th European Congress ERTS Embedded Real Time Software*, Toulouse, France, 2008.

[14] D. Kästner, C. Rustemeier, U. Kiffmeier, D. Fleischer, S. Nenova, R. Heckmann, M. Schlickling en C. Ferdinand, „Model-Driven Code Generation and Analysis," in *SAE World Congress*, 2014.

[15] S. Salvi, D. Kästner, T. Bienmüller en C. Ferdinand, „Exploiting Synergies between Static Analysis and Model-Based Testing," in *Proceedings of the 11th European Dependable*

Computing Conference (EDCC '15), 2015.

[16] AbsInt Angewandte Informatik GmbH, „XTC Language Specification Version 2.4,” 2016. [Online]. Available: http://www.absint.com/xtc/.