ASSUME

ITEA3

Affordable Safe & Secure Mobility Evolution

ASSUME Standardization Activities

Deliverable D6.7.1

Deliverable Information			
Nature	Document	**Dissemination Level**	Public
Project	ASSUME	**Project Number**	14014
Deliverable ID	D6.7.1	**Date**	29.08.2018
Status	Final	**Version**	1.0
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
Heiko Dörr	MES	doer@model-engineerings.com
Matthias Kern	FZI	matthias.kern@fzi.de
Frédéric Loiret	KTH	floiret@kth.se
Thomas Peikenkamp	OFFIS	thomas.peikenkamp@offis.de
Philipp Reinkemeier	OFFIS	philipp.reinkemeier@offis.de
Arturo Tejada Ruiz	TNO	arturo.tejadaruiz@tno.nl
Tino Teige	BTC ES	tino.teige@btc-es.de
Bernard Schmidt	BOSCH	bernard.schmidt@de.bosch.com
Carsten Sinz	KIT	carsten.sinz@kit.edu
Ferhat Erata	UNIT	ferhat@computer.org
Björn Koopmann	OFFIS	bjoern.koopmann@offis.de

Change and Revision History

Version	Date	Reason for Change	Affected sections
0.1	09.05.2018	Initial version	All
0.2	15.05.2018	SUP & Analysis Methods	3.1, 3.2
0.3	11.09.2018	Description of the ASC3F format	4
1.0	13.09.2018	Final version	All

Table of Contents

1. Executive Summary

In order to consider and keep up with the up-to-date science and technology, related work and tools are analyzed in each technical work package. The topic of this deliverable are the standardization activities of the ASSUME project. This includes collecting the standards that are used by the ASSUME project in the different work packages, as well as how the ASSUME project contributes to existing standards or creates new ones.

2. Overview

In summary, we have identified three main areas where the ASSUME projects builds upon existing standards, extending them, as shown in Figure 1: formalization of requirements, exchange of results of analysis tools, and tool and data interoperability.

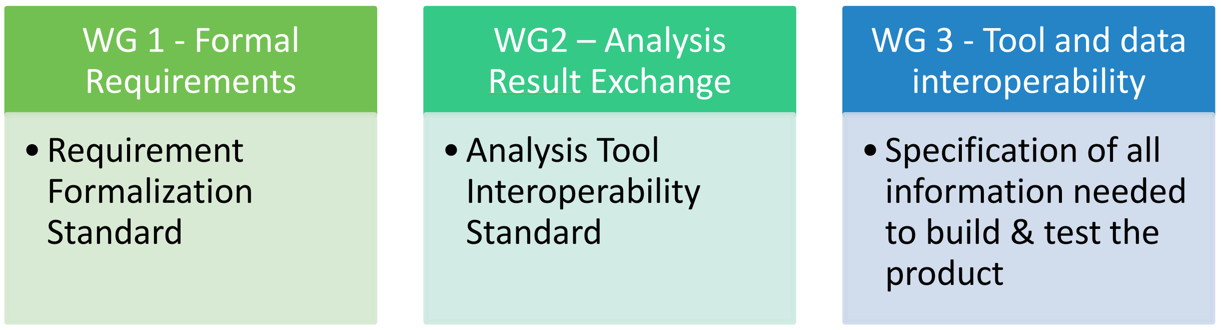

WG 1 - Formal Requirements
• Requirement Formalization Standard
WG2 – Analysis Result Exchange
• Analysis Tool Interoperability Standard
WG 3 - Tool and data interoperability
• Specification of all information needed to build & test the product

Figure 1: Main Areas of ASSUME Standardization Activities

These standardization activities are mainly driven by the work packages 3 (System Engineering Methodology), 2 (Scalable Zero-Defect Analysis for Single Core) and 5 (Scalable Zero-Defect Analysis for Multi-Core). The standardization activities corresponding to these areas will be reported in the subsequent sections of this document.

3. Formal Requirements

Guaranteeing the validity of requirements in a safety-critical context like in automobiles or airplanes is of utmost importance as failures of such systems often lead to life-threatening situations. Checking that a system under test meets a set of requirements can be performed at each level of the development, i.e., from the very beginning where just the natural-language requirements are known, over the stages where a system model, a system implementation, and a hardware virtualization exist, up to the final product. Since the process of requirements checking is time-consuming and expensive, in particular, if malfunction is detected at some very late point, there is an industrial trend to find system failures very early and preferably in an automatic way. There exist a wide variety of solutions pioneered by academia, commonly referred to as formal methods, being able to support this trend from a technological point of view. The off-the-shelf application of formal methods in industry however is at an early stage due to the lack of (de facto) standards and easy-to-use engineering tools. There are two main challenges to advise non-expert industrial users to deploy the power of formal methods in their projects and applications: first, providing intuitive description languages to formalize requirements given in natural language, and, second, efficient methods to achieve requirements integrity at each stage of the production, preferably just pushing a button.

To tackle these challenges, the ASSUME project also works on requirement specification languages that are close to natural language, but having formal semantics, thus supporting the transition from informal to formalized requirements. Once formalized, the requirements can be interpreted by corresponding tools, which, for instance, allows automatic generation of test-cases or checking for consistency of requirements.

3.1. Requirement Specification Languages

This section is on the Simplified Universal Pattern (SUP) that is one of the formal requirement specification languages which have been used and/or developed in the ASSUME project.

Description languages for formalizing natural language requirements should be as intuitive as possible, easy to understand, and preferably presented in a graphical way such that formalization of human-readable to machine-readable requirements becomes a common engineering task without being very prone to errors. BTC-ES devised such a specification method called *simplified universal pattern* (SUP) which is implemented in its product BTC EmbeddedPlatform®. The SUP approach is based on the observation that the clear majority of real-life safety-critical requirements for components can be expressed by temporal trigger/action relationships such as the textual requirement, “*If the driver up or passenger up switch is pressed then the window has to start moving up in less than 50 ms*”. A SUP explicitly introduces artefacts like trigger and action to close the gap between human intuition of a requirement and its formalized description; i.e., artefacts that a requirement engineer talks about are directly reflected in the specification formalism, as shown in the following figure. We remark that a trigger or an action itself is not limited to be instantaneous but can have a temporal extent.

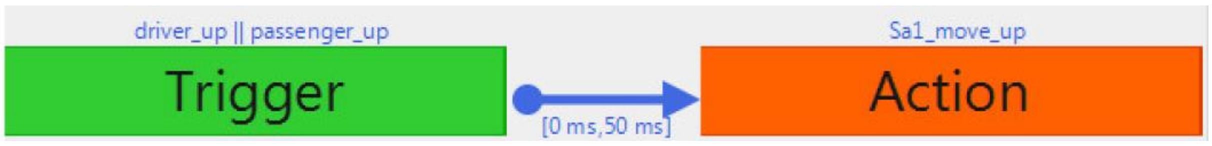
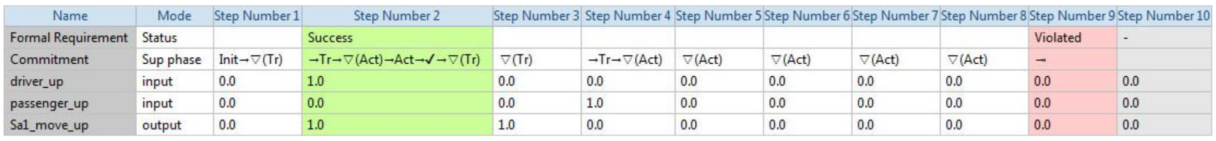


Figure 2: Graphical Representation of an SUP

In brief, the semantics of a SUP is as follows. A trigger or action is started by consuming its start event and successfully passed by accepting its end event in the specified time interval, while its condition must hold in between. A trigger or action fails during processing if its condition became false or its end event was not observed in the time interval. A SUP is successfully passed by an execution if its trigger and action is successfully passed by this execution and their temporal relation is met. A SUP is violated if there is some execution passing the trigger, for which the action does not start in the specified time interval or for which the action fails after entering it.

For a brief example, consider the SUP from the figure above. Please note that whenever a trigger or action is an instantaneous event then their respective start event, condition, and end event are equal. In such cases, only the condition is depicted in the graphical SUP description for the sake of clarity, compactness, and usability. One possible SUP execution is shown in the following figure.



Name	Mode	Step Number 1	Step Number 2	Step Number 3	Step Number 4	Step Number 5	Step Number 6	Step Number 7	Step Number 8	Step Number 9	Step Number 10
Formal Requirement	Status		Success							Violated	-
Commitment	Sup phase	Init→▽(Tr)	→Tr→▽(Act)→Act→✓→▽(Tr)	▽(Tr)	→Tr→▽(Act)	▽(Act)	▽(Act)	▽(Act)	▽(Act)	→	
driver_up	input	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
passenger_up	input	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0
Sa1_move_up	output	0.0	1.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0

Figure 3: Possible SUP Execution

In step 2 the expression of the trigger condition `driver_up || passenger_up` holds as `driver_up` is true, and thus the trigger is passed. The SUP is then ready to observe the action which happens immediately as `Sa1_move_up` is also true in step 2. The SUP is successfully passed and waits for a new trigger. The next trigger is consumed in step 4 due to `passenger_up`. Since the expression of the action condition `Sa1_move_up` does not hold in the following 5 steps/50 ms (where one step corresponds to 10 ms), the SUP is violated in step 9.

3.2. Requirement Analysis Methods

This section is on requirement analysis methods which have been used and/or developed in the ASSUME project.

BTC-ES connects its specification language SUP with several applications and use cases in BTC EmbeddedPlatform® to generate added values of specification efforts made by users.

Formal Test. Validating requirements by means of testing is a well-established industrial process and always applicable if the system under test can be simulated. BTC EmbeddedPlatform® supports automatic validation of SUPs wrt. test executions resulting from Simulink®, TargetLink®, or production code as well as for imported test executions from external architectures, with dedicated test reporting for all considered system architectures.

Although extensive and sophisticated tests passed on model and code level are a convincing argument for system safety, the system behavior still is uncertain in its intended embedding (hardware) environment. To achieve more confidence in a more realistic setting, dSPACE offers a wide range of hardware-in-the-loop (HIL) simulators for processing large-scale Simulink® models in real-time and in multi/many-core environments. BTC EmbeddedPlatform® provides a technical solution to integrate SUP specifications into HIL systems with the objective of automatic online testing of formalized requirements. It is worth to mention that in contrast to the “ideal” model view, within the real-time testing phase, new aspects play an important role such as timing tolerances for handling permissible computation and communication delays being easily expressible in SUP by its nature.

We finally mention the use case of self-monitoring which goes one step further: a code implementation of the specification probes the final product, e.g. a car, by monitoring the correctness of the system during lifetime. BTC EmbeddedPlatform® provides source-code export of SUP specifications to support this use case in principle.

Debugging. Counterexamples, i.e., system executions violating the specification, are very precious in order to understand and eliminate possible malfunctions. Owing to its temporal nature, counterexamples of SUPs can be illustrated in a very lucid and comprehensible way, linking each execution step to the current status of the SUP as shown in Figure 3.

Requirement Coverage. If formal testing of all requirements was successful, i.e., all SUPs are not violated by the existing test suite, it often remains the question of how comprehensive the requirements are covered by the test suite. The system execution from Figure 3 covers two test scenarios, namely the one where `driver_up` holds but not `passenger_up` and where `passenger_up` holds but not `driver_up` (the latter actually revealing a requirement violation). The test where both signals are true however is not covered. Note that a test scenario where the trigger condition is false is pointless in the sense that the entire SUP is not touched. Thus, a useful notion of requirement coverage should deal with such “interesting” test cases and give a reliable measure of how good the requirements are covered by the test suite.

BTC EmbeddedPlatform® offers the notion of trigger coverage. Intuitively, we want to measure how many different ways a trigger can be processed were actually seen by the formal test. More precisely, from the expressions of the trigger events and condition, new goal expressions are derived that encode “interesting” waypoints through the trigger as indicated in the following figure.

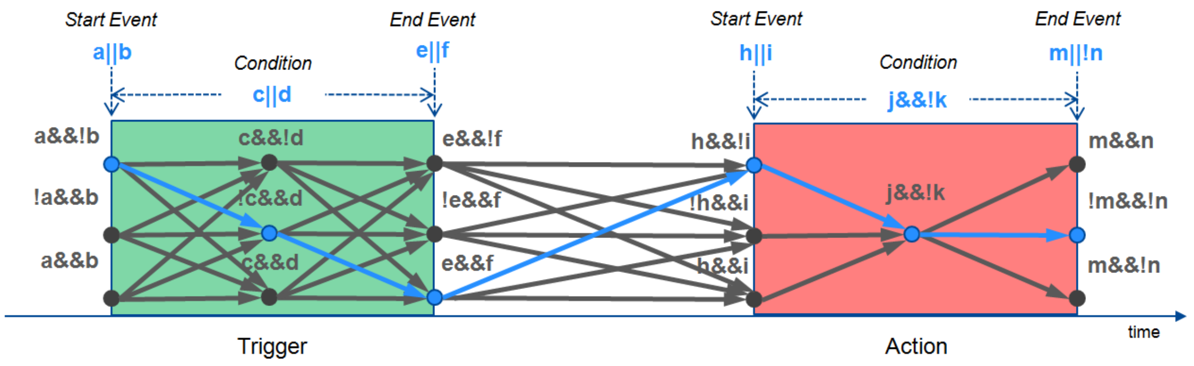


Figure 4: Example of a Covering Execution

For instance, from the trigger start event `a||b` the goal expressions `a&&!b`, `!a&&b`, and `a&&b` are derived. Trigger coverage then aims at visiting each of the waypoints of the trigger while afterwards successfully passing (or violating) the SUP. An example of a covering execution is shown in blue in the figure above.

Formal Verification. A significant test suite that fully covers the safety-critical requirements both on model and code level is an important achievement and boosts confidence of the product quality. It is, however, clear that testing cannot be exhaustive due to the fact that system errors may remain undetected even for only rare occasions. To detect such remaining erroneous behavior the system can be exhaustively checked against specification by using so-called model checking techniques. In case an error was found by model checking, a counterexample is generated supporting the debugging process. Although model checking is a very active research area in academia and also successfully applied in industry with increasing frequency, it must be often tailored to the corresponding application, domain, or use case for the sake of efficacy and efficiency. BTC EmbeddedPlatform® provides model checking technology that is particularly tailored to check TargetLink®-generated production C code against SUP specifications.

Test Case Generation. Finding test cases that yield a convincing requirement coverage most often is a manual and hence very time-consuming task. Moreover, if some test goals are not covered by the test suite then it is not always clear whether the test goals can be reached at all or whether some test cases are missing. BTC EmbeddedPlatform® automatically generates test cases for requirements with the objective of achieving full requirement coverage. Since model checking techniques are employed, test goals can also be certified as unreachable. Test case generation from requirements can be performed at a very early level, namely when just the formalized SUP specifications are known, or at a level where the system implementation is already present.

Consistency Analysis. In early design phases, formalized requirements shall be checked for consistency. Intuitively, a consistency analysis discovers conflicts between requirements that prevent the specification from being implementable. For the SUP, we propose the use of formal methods here. The following formal definitions of consistency have been prototypically implemented for the SUP on top of state-of-the-art model checkers. The most basic definition of consistency is basically *existential consistency* [1]: A set of requirements is existentially consistent if there exists at least one execution that satisfies all the requirements. Triggered existential consistency extends this notion by requiring that in this satisfying execution also every trigg er of a formal requirement is seen at most once. This is further extended in [2]: Here, the testcase generation feature by BTC EmbeddedPlatform® described above is used. A set of requirements is considered consistent if a test coverage of 100% (wrt. some coverage metric) is possible. The coverage metrics used are the same as above. As another enhancement of existential consistency, *partial consistency* [3] has been developed in ASSUME. For partial consistency the temporal constraints in SUP instances are analyzed for critical cases. Such a critical case might be if triggers of different requirements occur simultaneously. Requirements are partially consistent if, for every identified critical case, an execution exists where all requirements are still satisfied.

4. Analysis Result Exchange

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 claims, for instance, “Safety requirements shall be traceable...” ([4], 6.4.3.2) or “The traceability of safety-related hardware elements shall be ensured,...”([4], 7.4.5.3). Furthermore, the traceability between design, implementation, validation and test results should be possible to ensure that all data is valid in every point in time. In the context of static code analysis, traceability should be achieved between the analysis results of the tool and the corresponding software components, model artefacts such as Matlab/Simulink or Enterprise Architect, and requirements.

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 suitable to combine various tools having different strengths.

To this end, first, the static analysis tools’ outputs (e.g., reported potential errors) have to be comparable, i.e., the reported error categories have to be standardized. Secondly, also the input to the tools should be defined only at a central location once, to be able to compare the verification results in a meaningful way. This is currently difficult to achieve since there is no standardized interchange format for static code analysis tools. Actually, there is a need for two interchange formats: one for the analysis results and one for the configuration that describes the target system. Typical inputs are, for example, environment configurations and used platform. The standardized input and output enable usage of these tools in different stages of the development flow. It also allows combination of different checks, e.g., when uncertain results of a first check are proven in a second check with a different tool or approach.

4.1. ASSUME Static Code Analysis Tool Common Configuration Format

The ASSUME Static Code Analysis (SCA) Tool Common Configuration Format (ASC3F) serves two purposes: defining a common file format to specify program verification tasks, as well as the assumptions underlying the verification, and, defining a common file format to specify program verification results. The format is laid down in the form of XML Schema Definitions (XSD) with additional semantical annotations. Having a common exchange format simplifies both cooperation between different tools as well as integrating tools into a larger tooling workflow.

To this end, the **configuration format** must allow the specification of analysis tasks as independent from the specifics of individual tools as possible. For example, a common mean of configuring the interpretation of programs needs to be established. Furthermore, the SCA tool configuration should facilitate the specification of machine-independent configurations, e.g., hiding differences in the concrete paths of source code files. As a secondary goal, the configuration should be specified in a modular way, facilitating reuse of configurations.

With regards to the **report format**, we aim to establish a common hierarchy of check categories with accompanying check semantics, making results obtained from different tools directly comparable. Finally, the format needs to be extensible, allowing the addition of language support and analysis feature configuration later.

For the format design, we therefore use a set of guiding policies:

- Common semantics, e.g. for checks, should be defined as needed, but based on the semantics used by existing SCA tools where no conflicts arise.
- The format may allow tool-dependent configuration, but the amount of necessary tool-dependent configuration needs to be minimized. Users should be able to perform analysis tasks without any tool-specific configuration at all.
- The format may allow machine-dependent specification of paths but needs to provide means to the user to create configurations where any machine-dependent settings are as isolated from the rest of the configuration as possible.
- The format should consist of lightweight, reusable structures. Language-dependent structures should be isolated and their integration into the format must allow the addition of new language-dependent structures without breaking backward compatibility.
- Configurations should be composed of small, reusable and extensible parts, which may be aggregated to form a complete SCA tool configuration.

The details of ASC3F are specified in Appendix A.

4.2. The SARIF Format

Over the course of the ASSUME project, another format for exchanging results from static analysis tools emerged, mainly driven by Microsoft: The Static Analysis Results Interchange Format[1] (SARIF), which defines a standard format for the output of static analysis tools. However, compared to the ASSUME exchange format, it only covers some aspects mainly dealing with tool results, but not with tool setup. The SARIF format is already well-documented, possesses a mature language design and comes with an SDK to facilitate its use. The format's maintainers just initiated a standardization initiative[2] through OASIS (https://www.oasis-open.org).

We started to examine a unification of the two formats by performing a gap analysis and by contacting the initiators of the SARIF format. This work is still ongoing. The preliminary results of the gap analysis are summarized in the following illustration:

[1] https://sarifweb.azurewebsites.net

[2] https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=sarif

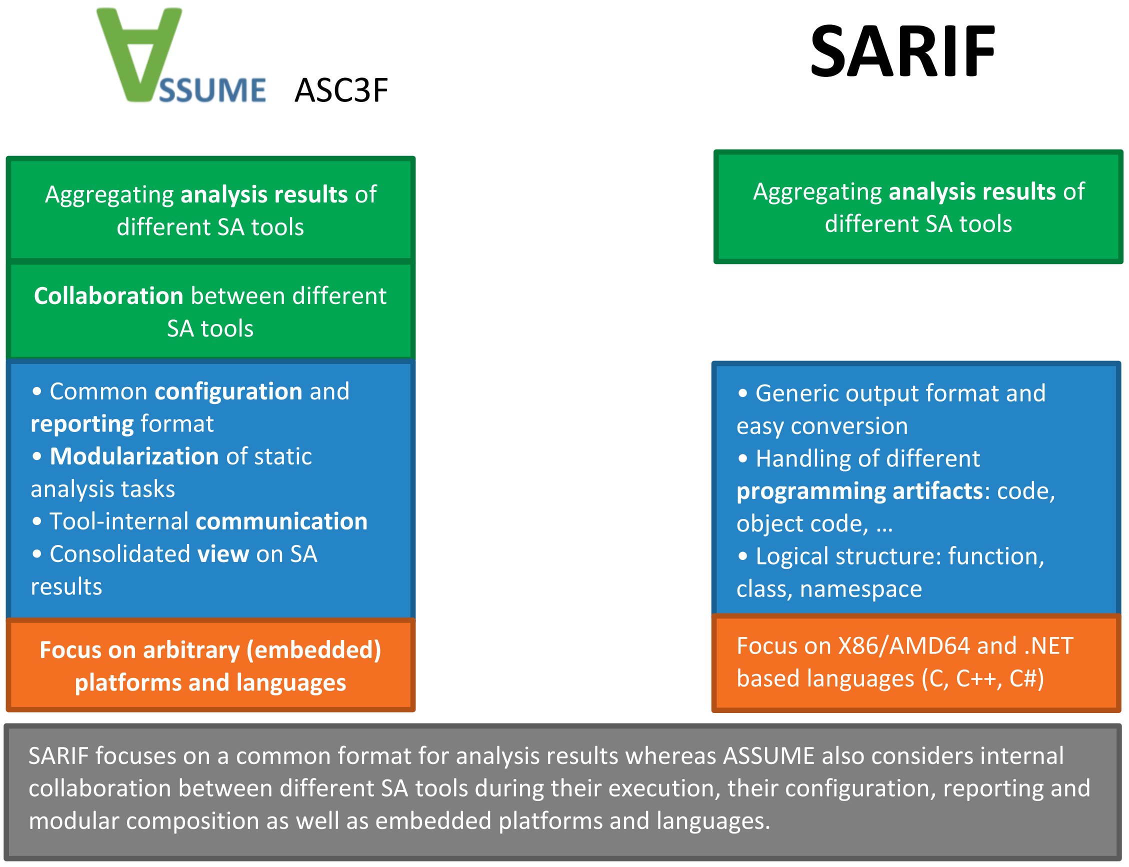

SSUME ASC3F
SARIF
Aggregating **analysis results** of different SA tools
Collaboration between different SA tools
• Common **configuration** and **reporting** format
• **Modularization** of static analysis tasks
• Tool-internal **communication**
• Consolidated **view** on SA results
Focus on arbitrary (embedded) platforms and languages
Aggregating **analysis results** of different SA tools
• Generic output format and easy conversion
• Handling of different **programming artifacts**: code, object code, …
• Logical structure: function, class, namespace
Focus on X86/AMD64 and .NET based languages (C, C++, C#)
SARIF focuses on a common format for analysis results whereas ASSUME also considers internal collaboration between different SA tools during their execution, their configuration, reporting and modular composition as well as embedded platforms and languages.

Figure 5: Comparison of ASC3F and SARIF

5. Tool and Data Interoperability

The heterogeneity and complexity of modern industrial products requires the use of many engineering software tools, needed by the different engineering disciplines (such as mechanical, electrical, embedded systems and software engineering), and throughout the entire development life cycle (requirements analysis, design, verification and validation, etc.). Unless interoperability mechanisms are developed to connect information across the engineering tools, one may well end up with isolated “islands of information”, given the natural distribution of information across the many tools and data sources involved. As an example from the automotive industry, the functional safety standard ISO 26262:2011 [4] mandates that requirements and design components are developed at several levels of abstraction; and that a clear traceability exists between requirements from the different levels, as well as between requirements and system components. The earlier practice, in which development artefacts are handled as text-based documentation, rendered such traceability ineffective – if not impossible. Even with the adoption of model-driven engineering, it remains a challenge to trace between the artefacts being created by the various engineering tools, to comply with the standard.

In summary, current development practices need a faster shift from the localized document-based handling of artefacts, towards a Federated Information-based Development Environment (F-IDE), where the information from all development artifacts is made accessible, consistent and correct throughout the development phases, disciplines and tools.

5.1. Interoperability of Safety Engineering Tools and Architecture and Design Tools

In safety-critical domains such as automotive, railway, and avionics, even a small failure of a system might cause injury to or death of people. Several international safety standards are introduced as guidelines for system suppliers to keep the risk of systems at an acceptable level, such as IEC 61508 (multiple domains), ISO 26262 (automotive domain), DO 178C (avionic domain), CENELEC railway standards (railway domain). In the automotive domain, currently the ISO 26262 standard, which is a goal-oriented standard for safety-critical systems within the domain of road vehicles, is the state of the art. This is, of course, the applicable standard for the Dutch (VDL) use case, which is the driver for TNO’s developments in the ASSUME project.

After its introduction in 2011, ISO 26262 has attracted more and more attentions in the automotive domain. There are more than 120 work products generated through out safety lifecycle suggested by this standard. This makes managing traceability and consistently of the information an absolutely necessity for assuring safety and compliance. In order to be able to maintain the abovementioned traceability when designing a vehicular system, it is important to have a well-structured process in place. This is already partly ensured when ISO 26262 is followed, however the norm still leaves a lot open aspects for interpretation and itself cannot guarantee quality of a design process.

TNO uses an existing commercial tool called Enterprise Architect (EA) to perform system design and analysis in a structured way. To achieve this, a specific “way of working” was set up by customizing the EA user interface and by connecting it with several other software tools (e.g., Excel, PLAATO, etc.). Enterprise Architect supports design and architecture of a system at several levels of abstraction. It also supports traceability to documentation, code simulation and centralization of the design. One way to ensure that traceability is maintained is to make it very easy to connect design diagrams and documentation to the development software and developed code. EA contains native solutions to link to documentation and developed code.

By making use of different diagrams that contain specific information for specific people, a natural layered and ordered representation of a system design can be presented to the user. This together with EA's tools for maintaining traceability ensures that documentation about the design, the design decisions, and other project information is kept very close to the actual system development.

Within the ASSUME project, TNO has made an extension to EA that enables a specific form of analysis once the right information is entered into a database. This database is typically filled by drawing diagrams with elements in EA that contain specific information about the system. This information is supplied by the system designers. A very important aspect of the system architecting tool is that it can be used from the start of the project and it acts as a central reference point to the latest developments done throughout a project. This also makes sure that the tool and the diagrams including all information are used with the latest state of the architecture.

As mentioned above, our development enables the user to deliver design input through a graphical representation of the system. To this end, three diagram templates have been added to those that are already available in Enterprise Architect:

- The first diagram is a function description diagram that shows which functions are present in the system. It captures information about what these functions do and the specific details about the information that is exchanged between these functions.
- The second diagram is a hardware description diagram that captures details on how hardware is interconnected and what properties the hardware has. It also defines the interfaces that are used to communicate between hardware components. Next to that also properties required for fault tree generation (e.g., failure probability or failure rate).
- The third diagram is the deployment diagram that contains information on how functions are deployed or mapped to the hardware. In this way, an engineer can conduct experiments with deploying multiple functions on one or more Electronic Control Units (ECUs) or shift tasks between ECUs.

When all information is correctly entered, it can be exported to Matlab for further processing. This is done by a plug-in of EA developed by TNO, which exports an ".m" file as input for Matlab. This file is then executed, yielding a representation in Matlab of the system deployment.

This representation allows the user to perform system analysis using a custom-made Matlab graphical user interface (GUI). This GUI can be used to generate a fault tree using the information that was entered in the model diagrams. When information is missing, the tool will try to guide the user to enter all required information in EA. Once everything is correct, the fault tree can be analyzed structurally as well as quantitatively. The structural analysis can be performed from a very early stage in the project as numbers are not strictly required only the structure of the system will suffice. The quantitative analysis requires actual numbers such as failure rates or importance metrics of specific elements in the system. The GUI offers the user several options for both forms of analysis, however specific knowledge of the analysis methods is required to be able to interpret the results.

For structural analysis the tool offers the ability to look at:

- Modules in the tree. These are independent regions which will show how a certain part of the tree is independent of other parts of the tree. This can indicate containment regions, in which failures will not propagate to other branches of the tree. These modular regions can be emphasized by the tool.
- Minimal cut-sets. These are the smallest terms of the logical formula (Boolean expression) that corresponds with a tree. The expression describes how the Basic Events contribute to failure of the top node of the tree. Basic events are considered to be the smallest elements that can fail in the system. By looking at the expressions one gets insight in which events may result in system failures, if there are only few (or only one), it means that few faults are required to trigger a system failure.
- The actual tree and structure of the tree can be seen as making a graphical representation. This will allow the analyst to visually inspect the tree.

For quantitative analysis the tool offers the ability to look at:

- Failure probabilities of the basic events and how these are distributed with respect to each other.
- Failure probabilities of the intermediate events and top event. These probabilities have to be calculated by finding the probability expression from the logical formula. The tool can perform these calculations.
- Importance metrics. The tool offers the ability to calculate importance metrics that will give information about the relative importance of the basic events in the tree. The tool supports the Birnbaum and the Fussel-Vesely importance measures.

The Matlab tool allows the user to store results in “.mat” files to be used for later analysis. Node properties can be adapted with the tool once the tree is generated. Additionally, one can build fault-trees directly in EA. A tree can then be exported to an “.m” file and be analyzed in the tool. This way the fault-tree generation step can be skipped.

5.2. Interoperable Toolchain for Requirements-Driven Model-Based Development

In this section we present a sample toolchain that provides solutions for ensuring consistency, correctness, model quality, centralized quality monitoring, and partially traceability. The toolchain has been evaluated on an example from automotive industry that has been initially developed in the SPES_XT project and has been extended with code generation to fit the needs of ASSUME. A more comprehensive description of the toolchain, with more details about the tool functionalities, can be found in [5]. Although the running example has low safety criticality, the methods for analysis and test demonstrated in the toolchain are usually applied to safety critical systems. They are recommended in ISO 26262 for any element that needs compliance with the standard.

The example represents an automotive adaptive light system (AL) which contains functionality such as adaptive high and low beam, turn signaling, cornering lightning, and ambient lightning. The functionality chosen to elaborate the interoperable toolchain is the cornering light. Cornering light is illuminating the area ahead and to the side of the vehicle to look around the bend. This functionality is activated when the indicator or turn the steering wheel is operated during night-time driving.

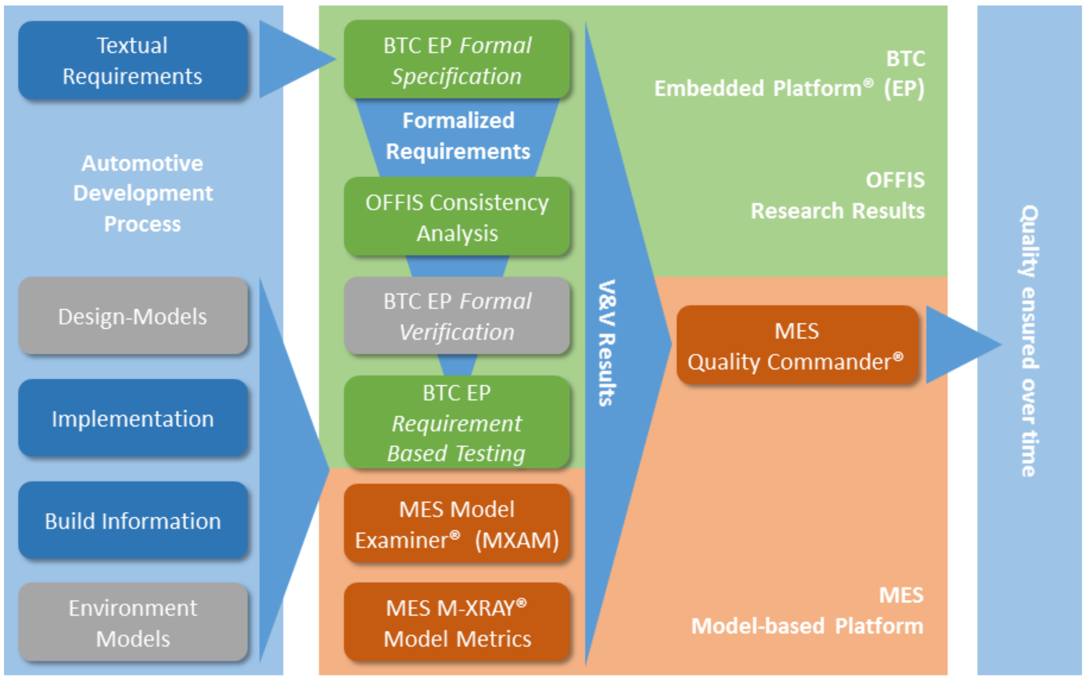

Textual Requirements
Automotive Development Process
Design-Models
Implementation
Build Information
Environment Models
BTC EP *Formal Specification*
Formalized Requirements
OFFIS Consistency Analysis
BTC EP *Formal Verification*
BTC EP *Requirement Based Testing*
MES Model Examiner® (MXAM)
MES M-XRAY® Model Metrics
V&V Results
BTC Embedded Platform® (EP)
OFFIS Research Results
MES Quality Commander®
MES Model-based Platform
Quality ensured over time

Figure 6: Overview of the Collaborative Toolchain from a Process Perspective

In general, the functionality represented by the running example is highly distributed on various ECUs that communicate via the car's vehicle busses, such as CAN, LIN, and Automotive-Ethernet in an AUTOSAR environment. The running example abstracts this environment, as in context of the interoperable toolchain, the textual (informal) requirements and model representation are in focus. Containing 113 selected functional and non-functional requirements it is still big enough to show up realistic challenges. The implementation is done using the model-based development tools Simulink® and TargetLink® where TargetLink® is used to generate ANSI-C source code.

An overview of the collaborative toolchain is provided above. The workflow starts with textual requirements describing the cornering light. Implementation (Simulink®) and build information to generate ANSI-C code are initially present in our running example. In a real-world workflow, design and environment models would be used as well. In the example we start directly with an implementation model. Since the example system has little interaction with the environment, we include the most important environment constraints (e.g., value ranges) in the specification but do not model the environment's behavior. As we did not use design and implementation models in the case study, they have been greyed-out in the figure. The textual requirements are formalized within BTC EmbeddedPlatform® (BTC EP) by using the graphical specification language *simplified universal pattern* (see Section **Fehler! Verweisquelle konnte nicht gefunden werden.**). Then the formal requirements are checked for consistency, and afterwards the implementation is verified against them using requirements-based testing of BTC EP. Since we omit a design model in our example, we run MIL and SIL tests directly against the implementation model. The inputs of the system are fully defined in the test cases, therefore we can run the tests without implementing an environment model. Using generated test cases, these are correct by construction with respect to the formal requirements. In a complete workflow, the correctness of design-models could be proven by the formal verification feature of BTC EP.

Besides checking correctness and consistency, we evaluate the model quality using guidelines and metrics in MES Model Examiner® and MES M-XRAY®. Results from all steps are collected and displayed with the help of the MES Quality Commander® which tracks quality over all revisions of the development cycles. The different tools used in the collaborative toolchain are integrated to enable an information exchange for a better analysis and presentation of test results.

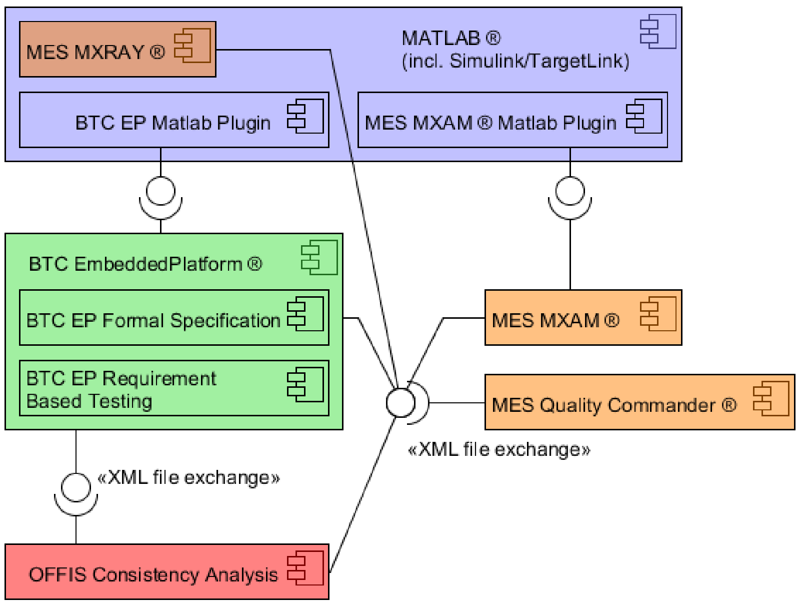


Figure 7: Interfaces Between the Tools

The component diagram above shows which tools have interfaces to each other. The data between tools is exchanged as XML files as depicted by the lollypop notation in the diagram: The consistency analysis consumes XML files from BTC EP; MES Quality Commander® consumes XML files from all the other tools in the chain. This reduces the overall manual effort of data maintenance to a one-click file export and import. Formal specification and requirement based testing are part of one tool, BTC EP, and hence share the same database. The BTC EP integrates with Simulink/TargetLink via a plugin in the MATLAB environment – the communication between BTC EP and MATLAB is fully automated and hidden from the user. There is no need for an interface between BTC EP and MES MXAM/M-XRAY® since the model quality analysis is independent from formal testing.

The XML formats used are natively supported formats of the industrial tools: The BTC EP provides the XML export out of the box. Also, the XML format for migrating quality data is natively used by MQC. Enough documentation for implementing the export/import functionality in the Consistency Analysis has been provided by the tool vendors. To avoid a semantical gap between the tools, the SUP semantics has been provided by BTC-ES to OFFIS in a more mathematical form compared to user documentation. This formal definition of the SUP semantics has been transformed into the internal representation of the consistency analysis prototype with a computer-aided and mathematically sound process.

Requirement quality has been the starting point for quality metrics being captured in the general exchange format for quality assessments. Further quality aspects along the line of development have been covered in and supported by the general exchange format. During the project, information provision has been realized for numerous tools spanning requirements quality analysis (e.g., consistency analysis), static analysis of models (e.g., MES Model Examiner, MES M-XRAY), or test tools (e.g., BTC Embedded Tester, MES Test Manager). All tools provide information on the quality of artefacts under investigation. These quality results will be collected, aggregated and visualized using the quality dashboard. During the project, the exchange format has been further evolved to meet the structures defined in ISO 25010 on quality measurements for software products. As a result of the project's research, the quality data exchange format will fit into the family of standard on quality measurement.

Implementation and evaluation of the tool chain showed that it is possible to use existing XML formats for data exchange between tools. Using formats already supported in some of the tools of the chain reduces development effort: There is no need to define a new exchange format and implementation is already done for one end of each tool connection. In case of our example toolchain all relevant data has been present in the exchange formats from the beginning; there was no need to extend the XML formats. However, in case of vendor specific formats the semantics must be communicated carefully.

5.3. Interoperability of Static Code Analysis Tools and Design Tools

The initiative Open Services for Lifecycle Collaboration (OSLC) offers specifications for a connection between tool data which addresses the application lifecycle management (ALM) and the product lifecycle management (PLM). There are implementations for OSLC like OSLC4J for Java. Furthermore, there is a model-based development tool, namely Lyo Modeler which is a tool created by KTH. The Lyo Modeler enables the model-based development of so-called adaptors. One Adaptor transfers the tool data in so-called resources, which are represented in the resource description format (RDF). RDF is a common format for Linked Data, where a resource called "subject" is linked via a property with another resource called "object". Each resource has a unified resource identifier (URI). The object itself can also be a subject in another context. With such triplets, it is possible to describe distributed and linked data. The RDF data are stored in a special data base called triple-store.

OSLC offers amongst other specifications for requirement-management, quality-management (QM), and architecture-management. With the architecture-management specification, data from modeling-tools can be mapped to resources. Data form a testing tool can be mapped with the help of the QM specification. However, a specification for the mapping of results from a static code-analysis-tool is missing. Therefore, a resource definition based on the QM specification and the ASC3F format was created. The use case for this resource definition is the connection of static code analysis-tools with other tools, so that all artefacts from the requirements, model artefacts, and generated source code, to the point of static code analysis results are traceable.

Even though the source code is typically complemented with the help of an integrated development environment (IDE), it is managed within a versions-management tool like Git. For an initial demonstration of this concept the connection between a version control system and a static code analysis tool was implemented. We assumed that the static code analysis tool delivers its results in the ASCF3 Format. For the implementation we used the static code analysis tool QPR-Refine and the version control system Git with GitLab. Furthermore, we used the Lyo Modeler for the implementation of the so-called Analysis Adaptor which offers the analysis resources and for the Code-Adaptor, which offers version-managed C-files from Git. Figure 8 depicts the resulting resource definition, which based on the OSLC QM specification as well as on the ASCF3 Format.

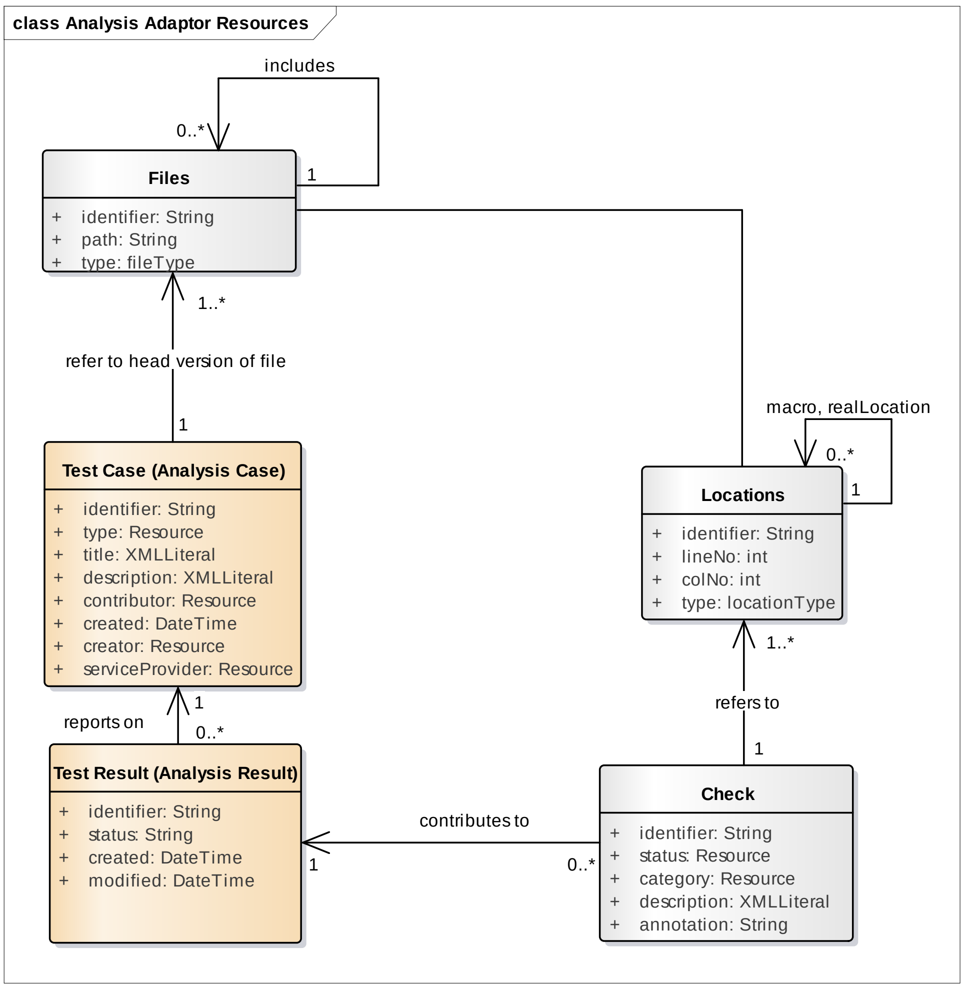


Figure 8: Resource Definition for the Analysis Adaptor

The orange boxes represent resources which are adapted from the OSLC QM specification. The gray boxes represent resources which based on the ASCF3 Format.

The new name of the adapted resources stands in brackets behind the original name from the OSLC QM specification. The Analysis Case is linked with all files which should be analyzed and contains the configuration for the static code analysis. The Analysis Result contains all so-called Checks of one analysis report. There can be different Analysis Results for different versions of the source code files. One Analysis Result refers to a specific Analysis Case.

Checks name the problems and are linked with the Locations where the problems occur. A Location gives the line and column number in the code where a problem was located. The Location is linked with the appropriate file, or with another location in the case of that the location is a macro location. Therefore, the Analysis Cases are traceable from its results up to the source code of a specific version. This approach offers potential for an automated analysis, similar to well-known continuous integration approaches and gives the possibility to compare different analysis results.

6. Standardization of Interoperability Specifications

Activities related to standardization of *Interoperability Specifications* within ASSUME have been mainly driven by KTH. In January 2018, KTH has officially launched the ICF[3] (the "*Interoperability Coordination Forum*"), supported by ARTEMIS-IA, which is chaired by Frédéric Loiret (WP3 leader of ASSUME) and Martin Törngren from KTH. The aim of the ICF is to foster the dissemination and industrial take-up of a set of interoperability specifications supporting the integration of IT and Systems development tools across various European programs (mainly around the ARTEMIS/ECSEL eco-system so far, but also with ITEA and H2020). During this first workshop, pre-standardization activities performed within the ASSUME data exchange format, namely ASCF3, has been presented. It is our objective to officially integrate these inputs into the ICF in the coming months, in order to increase the visibility of our ASSUME activities to a wider community. The ICF has been designed as a sustainable structure, funded by ARTEMIS-IA, and as part of its Standardization Working Group, we will therefore continue to promote ASSUME inputs on the long run.

KTH staff from the ASSUME project (Jad El-Khoury and Andrii Berezovskyi) are also actively involved in the OASIS standardization body, in particular, the OSLC set of standards which have been used as the cornerstone to ASSUME activities in the WP3. They are both active member of the OASIS Lifecycle Integration Core (OSLC Core) Technical Committee[4], and active member of the OASIS OSLC Domains Technical Committee (Domains TC)[5].

3 https://artemis-ia.eu/icf-workshop-2018.html

4 https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=oslc-core

5 https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=oslc-domains

7. Conclusion

This document reports on how standardization among system development tools has been achieved in the ASSUME project. In the ASSUME project standardization needs for requirement formalization, analysis result exchange, interoperability between tools and (configuration) data are addressed.

The developed toolchain (Section 5.2) demonstrates that the Simplified Universal Pattern (Section **Fehler! Verweisquelle konnte nicht gefunden werden.**) can be used to formalize functional requirements, such that different analysis tools interpret them with the same unambiguous semantics. Also, ASSUME shows how widely used industrial tools, such as Enterprise Architect (Section 5.1), can be extended to support an integrated development process. Additionally, to exploit data formats natively supported by third-party tools, a new exchange format (Section 0) for tool configuration and analysis results has been developed that facilitates tool interoperability. Adopting the exchange format for OSLC (Section 5.3) especially pushes tool integration and may be subject to standardization beyond ASSUME.

References

[1] 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, September 11-12, 2014. Proceedings*, 2014.

[2] T. Bienmüller, T. Teige, A. Eggers and M. Stasch, "Modeling Requirements for Quantitative Consistency Analysis and Automatic Test Case Generation," in *Workshop on Formal and Model-Driven Techniques for Developing Trustworthy Systems*, 2016.

[3] J. S. Becker, "Analyzing Consistency of Formal Requirements," in *Automated Verffication of Critical Systems, AVoCS 2018*, 2018.

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

[5] J. S. Becker, V. Bertram, T. Bienmüller, U. Brockmeyer, H. Dörr, T. Peikenkamp and T. Teige, "Interoperable Toolchain for Requirements-Driven Model-Based Development," in *ERTS 2018*, 2018.

Annex A: The ASC3F Format

This appendix presents the ASSUME static code analysis tool exchange format (ASC3F) as of August 2018.

ASC3F serves two purposes: Defining a common file format to specify program verification tasks, as well as the assumptions underlying the verification (Sec. A.1); and, defining a common file format to specify program verification results (Sec. A.2). The format is laid down in the form of XML Schema Definitions (XSD) with additional semantical annotations. The XML Schema Definitions will be made available for download.

A.1. Configuration

In this section, we present most aspects of the current state of the configuration format by example (see `ExampleConfiguration.xml` at the end of the appendix).

A.1.1. Top-Level Structure

A configuration consists of three major parts:

1. The common configuration (belonging to the global configuration) describing configuration items for all analysis tools and independent of machines executing the analysis tools;
2. the optional tool-specific configuration (also belonging to the global configuration) allowing users to configure analysis tool behavior which cannot currently be specified in the common configuration;
3. the local configuration, which is intended to allow users to specify, e.g., concrete directory replacements for the respective placeholders used in the other parts of the configuration.

For now, the configuration schema is designed with the usage of `XInclude` in mind: place the global and local configurations in separate files and create a configuration file by including both. Users then only need to supply a customized local configuration file.

```
<?xml version="1.0" encoding="UTF-8"?>
<ascccf:Configuration xmlns:ascccf="http://example.com/ascccf"
                      xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"
                      xsi:schemaLocation="http://example.com/ascccf
ASC3F.xsd">
  <ascccf:GlobalConfiguration>
    <ascccf:CommonConfiguration>
      <<Meta information>>
      <<Hardware targets>>
      <<Language targets>>
      <<Source code modules>>
      <<Execution model targets>>
      <<Check targets>>
      <<Analysis tasks>>
    </ascccf:CommonConfiguration>
    <<Tool-specific configuration>>
  <ascccf:GlobalConfiguration>
  <<Local configuration>>
</ascccf:Configuration>
```

A.1.2. Meta Information

Users can store a description of the configuration file in the metadata element. This element is optional.

```
<<Meta information>> =
  <Meta configurationName="Example configuration for ASSUME"
version="1.0"
        description="This is an example for the ASSUME configuration
format.">
    <Maintainer>Felix Kutzner (KIT)</Maintainer>
  </Meta>
```

A.1.3. Hardware Targets

To allow modular configuration files, most bits of the configuration are stored in entities identifiably by name, which can then be combined by to specify a concrete analysis configuration. *Hardware targets* are descriptions of the hardware on which the code to be analyzed is intended to run:

```
<<Hardware targets>> =
  <HardwareTargets>
    <HardwareTarget name="PPC32">
      <PhysicalHardwareTarget endianness="big"
                              unalignedDereferenceSupported="false"
                              pointerSize="32"
                              functionPointerSize="32"/>
    </HardwareTarget>
    <HardwareTarget name="x86">
      <PhysicalHardwareTarget endianness="small"
                              unalignedDereferenceSupported="false"
                              pointerSize="32"
                              functionPointerSize="32"/>
    </HardwareTarget>
  </HardwareTargets>
```

A.1.4. Language Targets

Language targets are named entities describing how to interpret source code of a given programming language. Currently, the format only supports C language targets. However, the configuration format is designed with extensibility in mind: new language targets, e.g. for C++, can be introduced at a later stage without affecting existing configuration files.

```
<<Language targets>> =
  <LanguageTargets>
    <LanguageTarget name="ISO C">
      <CLanguageTarget signedBitfields="true"
                       signedEnums="true"
                       fpRoundingMode="downward"
                       fpConstantRoundingMode="downward"
                       enumType="int"
                       inlineAssemblyHandlingMode="ignore"
                       standardRevision="ISO C">
        <Types>
          <Type name="char" size="8" alignment="8" atomic="false"/>
          <Type name="short" size="16" alignment="16" atomic="false"/>
          <Type name="int" size="32" alignment="32" atomic="false"/>
          <Type name="long" size="32" alignment="32" atomic="false"/>
          <Type name="long long" size="64" alignment="64"
atomic="false"/>
          <Type name="float" size="32" alignment="32" atomic="false"/>
          <Type name="double" size="64" alignment="64" atomic="false"/>
        </Types>
      </CLanguageTarget>
    </LanguageTarget>
```

A C language *subtarget* extends another C language (sub)target with preprocessor directives, include directories, and individual include files needing to be prepended to all C source code files which are analyzed using this C language subtarget. The latter may e.g. be used to include compiler-specific header files. Paths need to be specified as URIs. Within URIs, substrings matching `$_[A-Z]*_$` are placeholders for concrete paths specified in the local configuration. Include directories and files are used in the order of their appearance within the language targets. Via the insertionMode attribute, the user needs to specify whether the list of include directories resp. files needs to be prepended or appended to the subtarget parent's list of include directories resp. files (if applicable).

```
    <LanguageTarget name="ISO C with library includes">
      <CLanguageSubtarget superTarget="ISO C">
        <PreprocessorDirectives>
          <Definition identifier="STATIC_ANALYSIS"/>
        </PreprocessorDirectives>
        <IncludeDirectories insertionMode="append">
          <DirectoryURI>$_LIBCINCLUDES_$</DirectoryURI>
        </IncludeDirectories>
        <IncludeFiles insertionMode="prepend">
          <IncludeFile path="$_SYSINCLUDES_$/compiler_sys.h"
local="true"/>
        </IncludeFiles>
      </CLanguageSubtarget>
    </LanguageTarget>
```

The basic idea is to have a generic C language target and more concrete language targets via subtargets:

```
    <LanguageTarget name="LT for ExampleModule">
      <CLanguageSubtarget superTarget="ISO C with library includes">
        <IncludeDirectories insertionMode="append">
          <DirectoryURI>$_EXAMPLEMODULEPATH_$/include</DirectoryURI>
        </IncludeDirectories>
      </CLanguageSubtarget>
    </LanguageTarget>
  </LanguageTargets>
```

The LanguageTargets element is not the only place in which language subtargets may occur. For example, within source code modules, language subtargets may be specified for individual files (extending the language target which would otherwise be used to interpret that file).

A.1.5. Source Code Modules

Source code module elements are named entities describing sets of files needing to be analyzed. Users may specify an optional root URI, relative to which relative SourceFile URIs are interpreted. Within source code modules, source files are identifiable by an ID. Source code module elements may also contain information about which parts of the code needs to be stubbed.

```
<<Source code modules>> =
  <SourceModules>
    <SourceModule name="ExampleModule" rootUri="$_EXAMPLEMODULESRC_$">
      <SourceFiles>
        <SourceFile uri="main.c" id="1"/>
```

Needing further preprocessor directives and header files not visible to other source code files, the file dodgycode.c has an individual C language subtarget (extending the language target which would otherwise be used to interpret that file):

```
        <SourceFile uri="dodgycode.c" id="2">
          <LanguageTargetExtension>
            <CLanguageSubtarget>
              <PreprocessorDirectives>
                <Definition identifier="SILLY_ADD" expansion="x+y"/>
                <Definition identifier="KBD_PORT_ADDR" expansion="0x60"/>
                <Definition identifier="KBD_STAT_ADDR" expansion="0x64"/>
                <Definition identifier="KBD_CMD_ADDR" expansion="0x64"/>
              </PreprocessorDirectives>
              <IncludeFiles>
                <IncludeFile path="HideProprietaryCExtensions.h"
local="true"/>
```

```
          </IncludeFiles>
        </CLanguageSubtarget>
      </LanguageTargetExtension>
    </SourceFile>
    <SourceFile uri="shadycode.c" id="3"/>
  </SourceFiles>
```

Let's say, the module `ExampleModule` requires function stubs. Since function stubbing is language dependent, the format offers an optional `RequiredCStubs` element within source code modules. There are three ways of requiring a C function to be stubbed: requiring a *visibility controlled* stub means that the stub generator should create a skeleton stub for each visibility-controlling macro identifier specified in the `VisibilityControllingSymbols` list, controlling their visibility via corresponding `#ifdef` directives. Requiring a *universal* stub means requiring a single stub skeleton to be generated, which is not „guarded" by `#ifdef` directives. Furthermore, *autogen* stubs should not be implemented in source code files, but be generated on-the-fly be the analysis tool. (Note that the analysis tool only needs to interpret *autogen* stubs; the other information may be used for manual stub implementation or by stub skeleton generators.)
Stubbed functions are identified via URIs. For C, the namespace `cstub` is used, which is structured as follows:

- Functions *func* having external linkage are described by `cstub://globalscope/`*`func`*.
- Functions *func* having internal linkage for the file with ID *fileID* within source code module *module* are described by `cstub://filescope/`*`module`*`/`*`fileID`*`/`*`func`*.

Stubs for functions with external linkage may be assigned to groups. Stub generators should place all stubs of a group into the same file.

```
  <RequiredCStubs>
    <VisibilityControllingSymbols>
      <CVisibilityControllingSymbol name="ENABLE_LLBMC_STUBS"/>
      <CVisibilityControllingSymbol name="ENABLE_ASTREE_STUBS"/>
    </VisibilityControllingSymbols>
    <CUniversalStub uri="cstub://globalscope/open" group="posix_io"/>
    <CUniversalStub uri="cstub://globalscope/read" group="posix_io"/>
    <CUniversalStub uri="cstub://globalscope/write"
group="posix_io"/>
    <CVisibilityControlledStub uri="cstub://globalscope/fancy_rng"/>
    <CVisibilityControlledStub
        uri="cstub://filescope/ExampleModule/2/read_from_kbd"/>
    <CAutogenStub uri="cstub://globalscope/malloc"/>
    <CAutogenStub uri="cstub://globalscope/free"/>
  </RequiredCStubs>
</SourceModule>
```

For each stub URI, at most one corresponding stub entry may be present in a `RequiredCStubs` element.

A stub module generated using the stub specification given in ExampleModule might look like this:

```
    <SourceModule name="ExampleModule_stubs"
rootUri="$_EXAMPLEMODULESTUBS_$">
      <SourceFiles>
```

groups/posix_io.c implements all stubs of the group *posix_io*. All stubs implemented in a given source code file are listed using the ImplementsStubs element:

```
        <SourceFile uri="groups/posix_io.c" id="1">
          <ImplementsStubs>
            cstub://globalscope/open
            cstub://globalscope/read
            cstub://globalscope/write
          </ImplementsStubs>
        </SourceFile>
```

The stub *fancy_rng* was not assigned to a group, so it is placed in an individual file:

```
        <SourceFile uri="fancy_rng.c" id="2">

<ImplementsStubs>cstub://globalscope/fancy_rng</ImplementsStubs>
        </SourceFile>
```

Finally, the static function *read_from_kbd* has a location corresponding to its stub URI:

```
        <SourceFile uri="filescope/ExampleModule/2/read_from_device.c"
id="3">

<ImplementsStubs>cstub://filescope/ExampleModule/2/read_from_kbd
          </ImplementsStubs>
        </SourceFile>
      </SourceFiles>
    </SourceModule>
```

Having a module ExampleModule_stubs implementing the stubs for ExampleModule, we can create a third module composed of the former two. To complicate things, let's assume that our third module contains a file implementing the function *fancy_rng*, which is also implemented in the stub module.

```
<SourceModule name="ExampleModule_joined"
rootUri="$_EXAMPLEMODULEJOINED_$">
      <SourceFiles>
        <SourceFile uri="someOtherFancyRNG.c" id="1"/>
      </SourceFiles>
      <RequiresModules>
```

We include all files from ExampleModule:

```
        <RequiresModule name="ExampleModule"/>
```

We also include all of `ExampleModule_stubs`, however excluding all files implementing the stub `cstub://globalscope/fancy_rng`:

```
        <RequiresModule name="ExampleModule_stubs">
          <ExcludingFilesProvidingStub
uri="cstub://globalscope/fancy_rng"/>
        </RequiresModule>
```

Alternatively, we could have used an `<ExcludingFile module="..." id="..."/>` element to specify a file to be excluded from the inclusion. Since the required module may require further modules, it is necessary to specify the module name as well as the file's identifier within that module.

```
      </RequiresModules>
    </SourceModule>
  </SourceModules>
```

A.1.6. Execution Model Targets

Execution model targets are named entities specifying how software is executed, e.g., synchronous (only one thread running at the same time) vs. asynchronous (multi-threaded) execution and entry points:

```
<<Execution model targets>> =
  <ExecutionModelTargets>
    <ExecutionModelTarget name="ExampleExecModel">
      <CSynchronousCExecutionModeTarget>
        <EntryPoints>
          <EntryPoint>main</EntryPoint>
        </EntryPoints>
      </CSynchronousCExecutionModeTarget>
    </ExecutionModelTarget>
  </ExecutionModelTargets>
```

A.1.7. Check Targets

Check targets are named entities in which the user can configure requirements for checks, e.g. which runtime or MISRA checks need to be supported by the static analysis tool.

```
<<Check targets>> =
  <CheckTargets>
    <CheckTarget name="BasicChecks">
      <CCheckTarget>
        <RequiredRuntimeCheck>integer-arithmetic</RequiredRuntimeCheck>
        <RequiredRuntimeCheck>floating-point-
```

```
arithmetic</RequiredRuntimeCheck>
        <RequiredRuntimeCheck>data-flow</RequiredRuntimeCheck>
        <RequiredRuntimeCheck>control-flow</RequiredRuntimeCheck>
        <RequiredRuntimeCheck>memory-access</RequiredRuntimeCheck>
        <RequiredRuntimeCheck>c-assertions</RequiredRuntimeCheck>
      </CCheckTarget>
    </CheckTarget>
  </CheckTargets>
```

A.1.8. Analysis Tasks

Analysis tasks are named entities representing combinations of hardware targets, source code modules, language targets, check targets and execution model targets, thereby completing a configuration (modulo tool-specific settings). For example, to analyze the source code module `ExampleModule_joined` for x86 as well as for PPC processors, the user might create the following analysis tasks:

```
<<Analysis tasks>>=
  <AnalysisTasks>
    <AnalysisTask name="analyzeExampleModuleOnPPC"
                  hardwareTarget="PPC32"
                  sourceModule="ExampleModule_joined"
                  languageTarget="LT for ExampleModule"
                  checkTarget="BasicChecks"
                  executionModelTarget="ExampleExecModel"
                  missingRequiredCapabilityHandlingMode="warning">
      <ReportGeneratorConfiguration documentFormat="assume"/>
    </AnalysisTask>
    <AnalysisTask name="analyzeExampleModuleOnX86"
                  hardwareTarget="x86"
                  sourceModule="ExampleModule_joined"
                  languageTarget="LT for ExampleModule"
                  checkTarget="BasicChecks"
                  executionModelTarget="ExampleExecModel"
                  missingRequiredCapabilityHandlingMode="warning">
      <ReportGeneratorConfiguration documentFormat="assume"/>
    </AnalysisTask>
  </AnalysisTasks>
```

A.1.9. Tool-Specific Configuration

Some parts of the configuration are too tool-specific to be specified in the common configuration: for example, time- and memouts might be specified using different granularities, and some settings such as loop bounds are dependent on the fundamental approach of the analysis tool. Users may provide *tool configurations* containing basic parameters such as additional command line arguments for the analysis tool and language target extensions containing e.g. further C preprocessor directives (for a given source code file, the tool-specific language target extension extends the language target which would otherwise be used for the file. If the file has an individual language target, the tool-specific language target extends that target and is used instead).

```
<<Tool-specific configuration>>=
  <ascccf:ToolConfigurations>
    <ToolConfiguration name="LLBMC" customParameters="-x">
      <ForAnalysisTasks>
        <AnalysisTaskName>analyzeExampleModuleOnPPC</AnalysisTaskName>
        <AnalysisTaskName>analyzeExampleModuleOnX86</AnalysisTaskName>
      </ForAnalysisTasks>
      <LanguageTargetExtension>
        <CLanguageSubtarget>
          <PreprocessorDirectives>
            <Definition identifier="STATIC_ANALYSIS"/>
            <Definition identifier="ENABLE_LLBMC_STUBS"/>
          </PreprocessorDirectives>
        </CLanguageSubtarget>
      </LanguageTargetExtension>
```

The content of the `ToolSpecificConfiguration` element is not defined by this configuration format, but by the vendors of individual analysis tools. For example, in the case of LLBMC, such a `ToolSpecificConfiguration` might look like this:

```
      <ToolSpecificConfiguration>
        <LoopBound>40</LoopBound>
      </ToolSpecificConfiguration>
    </ToolConfiguration>
  </ascccf:ToolConfigurations>
```

A.1.10. Local Configuration

Finally, the *local configuration* contains replacement rules for URI substrings, which need to be applied to all URIs occuring in the global part of the configuration. For this configuration, a local configuration might have the following rules:

```
<<Local configuration>>=
  <ascccf:LocalConfiguration>
    <URISubstitutionRules>
      <URISubstitutionRule token="$_LIBCINCLUDES_$"
                           substitution="file:///usr/include/libc"/>
      <URISubstitutionRule token="$_SYSINCLUDES_$"
                           substitution="file:///usr/include"/>
      <URISubstitutionRule token="$_EXAMPLEMODULEPATH_$"
                           substitution="file:///ExampleProject"/>
      <URISubstitutionRule token="$_EXAMPLEMODULESRC_$"
                           substitution="$_EXAMPLEMODULEPATH_$/src"/>
      <URISubstitutionRule token="$_EXAMPLEMODULESTUBS_$"
```

```
substitution="$_EXAMPLEMODULEPATH_$/generatedStubs"/>
      <URISubstitutionRule token="$_EXAMPLEMODULEJOINED_$"
                           substitution="$_EXAMPLEMODULEPATH_$/joined"/>
    </URISubstitutionRules>
  </ascccf:LocalConfiguration>
```

A.2. Reports

In this section, we present most aspects of the current state of the report format (also by example).

A.2.1. Top-Level Structure

A report consists of six major parts:

1. a copy of the configuration used to obtain the results (see Section A.1),
2. a collection of execution reports,
3. a collection of source code file descriptions,
4. a collection of source code location descriptions,
5. a collection of check results,
6. and a collection of failure traces.

```
<?xml version="1.0" encoding="UTF-8"?>
<ascccf:Report xmlns:ascccf="http://example.com/ascccf"
               xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
               name="String"
               sourceVersion="String"
               xsi:schemaLocation="http://example.com/ascccf ASCCRF.xsd">
  <<Configuration>>
  <<Execution report collection>>
  <<Source file collection>>
  <<Source location collection>>
  <<Check result collection>>
  <<Failure trace collection>>
</ascccf:Report>
```

A.2.2. Execution Report Collection

An *execution report* is a named entity detailing which configuration has been used to configure the analysis tool and contains information about the analysis tool execution, such as warnings. Since the latter section remains to be designed, users can currently insert arbitrary XML data such as `<TODO_ConcreteExecutionReport/>` in its place.

<<Execution report collection>> =

```
  <ascccf:ExecutionReports>
    <ascccf:ExecutionReport
        name="Report for ExampleProject-on-PPC analysis"
        analysisTask="analyzeExampleModuleOnPPC"
        toolParameters="LLBMC"
        analysisBeginDate="2016-12-17T09:30:47Z"
        analysisFinishDate="2016-12-17T09:30:49Z"
        analysisComputerName="i11pc164"
        toolName="LLBMC 2016-12-10">
      <TODO_ConcreteExecutionReport/>
    </ascccf:ExecutionReport>
  </ascccf:ExecutionReports>
```

A.2.3. Source File Collection

To allow the identification of individual source files as well as e.g. include files not specified in source code modules, reports contain a separate collection of source file descriptions – a list of uniquely identifiable File elements. Where possible, these files are identified with their corresponding source-module-level descriptions via the module name and their ID within that module. Moreover, a hash sum can be stored for each source file. The source file descriptions contain language-dependent data: for example, a C source file may be flagged as preprocessed, while a C header file has an include directory attribute. Again, source file descriptions for other languages can be added without breaking backward compatibility.

```
<<Source file collection>> =
  <ascccf:SourceFiles>
    <File id="1">
      <CSourceFile path="$_EXAMPLEMODULESRC_$/main.c"
                   hashSum="1b826051506f463f07307598fcf12fd6"
preprocessed="false"
                   originModule="ExampleModule" idInOriginModule="1"/>
    </File>
    <File id="2">
    <CSourceFile path="$_EXAMPLEMODULESRC_$/dodgycode.c"
                 hashSum="3b5337aa426bb547efefb97edec54e3e"
preprocessed="false"
                 originModule="ExampleModule" idInOriginModule="2"/>
    </File>
    <File id="3">
      <CSourceFile path="$_EXAMPLEMODULESRC_$/shadycode.c"
                   hashSum="ff702f10bebfa2f1508deb475ded2d65"
preprocessed="false"
                   originModule="ExampleModule" idInOriginModule="3"/>
    </File>
    <File id="4">
      <CHeaderFile path="$_EXAMPLEMODULEPATH_$/include/ExampleModule.h"
                   hashSum="2f702f10bebfa2f1508deb475ded2d65"
```

```
preprocessed="false"
                   includeDirectory=""/>
    </File>
    <File id="5">
      <CHeaderFile path="compiler_sys.h"
hashSum="3f702f10bebfa2f1508deb475ded2d65"
                   preprocessed="false"
includeDirectory="$_SYSINCLUDES_$"/>
    </File>
    <File id="6">
      <CHeaderFile path="HideProprietaryCExtensions.h"
                   hashSum="4f702f10bebfa2f1508deb475ded2d65"
preprocessed="false"
                   includeDirectory="$_EXAMPLEMODULESRC_$"/>
    </File>
    <File id="10">
      <CSourceFile path="$_EXAMPLEMODULESTUBS_$/groups/posix_io.c"
                   hashSum="8633b81a334995b50b53df83581af093"
preprocessed="false"
                   originModule="ExampleModule_stubs"
idInOriginModule="1"/>
    </File>
    <File id="11">
      <CSourceFile path="$_EXAMPLEMODULESTUBS_$/fancy_rng.c"
                   hashSum="b06f74ff6378f4a2629621b3d8aa935f"
preprocessed="false"
                   originModule="ExampleModule_stubs"
idInOriginModule="2"/>
    </File>
    <File id="12">
      <CSourceFile
path="$_EXAMPLEMODULESTUBS_$/filescope/EM/4/read_from_device.c"
                   hashSum="c143a9ae806ab2c93ad4f4f593173bf0"
preprocessed="false"
                   originModule="ExampleModule_stubs"
idInOriginModule="3"/>
    </File>
    <File id="20">
      <CSourceFile path="$_EXAMPLEMODULEJOINED_$/someOtherFancyRNG.c"
                   hashSum="591d99a6a84b1e1dbb44395a3fa27d64"
preprocessed="false"
                   originModule="ExampleModule_joined"
idInOriginModule="1"/>
    </File>
```

In the future, the `SourceFiles` element will probably renamed to `SourceStorageCollection`, as source code may be stored outside of files, e.g., in preprocessor definitions:

```
    <File id="30">
      <CFilelessPreprocessorDefinition identifier="SILLY_ADD"
expansion="x+y"/>
    </File>
  </ascccf:SourceFiles>
```

A.2.4. Source Location Collection

The source location collection contains *locations* identifying places within the files described in the source file collection. The format supports multiple approaches of identifying such places: the C-specific one (including support for macro expansion) is recommended for locations in C source code. Additionally, support is provided for plaintext file/line/column and file/line location specifications. Finally, ranges in source code can be specified using a special location type.

```
<<Source location collection>> =
  <ascccf:Locations>
```

A basic C soure code location (without need for macro expansion) is given by a file/line/column triple:

```
    <Location id="1">
      <CRealLocation fileID="2" lineNo="22" colNo="8"/>
    </Location>
```

C macro expansions are represented by *macro locations*, consisting of a spelling location ID (the location within the macro definition) and the expansion location ID (the location where the macro is expanded).

```
    <Location id="10">
      <CRealLocation fileID="4" lineNo="10" colNo="12"/>
    </Location>
    <Location id="11">
      <CRealLocation fileID="3" lineNo="32" colNo="13"/>
    </Location>
    <Location id="12">
      <CMacroLocation spellingLocID="10" expansionLocID="11"/>
    </Location>
```

Ranges can be specified using a begin and an end location ID:

```
    <Location id="20">
      <CRealLocation fileID="2" lineNo="22" colNo="12"/>
    </Location>
    <Location id="21">
      <CRealLocation fileID="2" lineNo="22" colNo="20"/>
    </Location>
    <Location id="22">
```

```
      <RangeLocation beginLocID="20" endLocID="21"/>
    </Location>
```

Finally, locations can be specified in plaintext files:

```
    <Location id="30">
      <PlaintextRealLocation fileID="20" lineNo="5" colNo="53"/>
    </Location>
    <Location id="31">
      <PlaintextLineLocation fileID="20" lineNo="5"/>
    </Location>
  </ascccf:Locations>
```

A.2.5. Check Result Collection

The check result collection contains a Check element for each performed check. We plan to establish a common set of check categories (containing e.g. the category `integerarithmetic.divbyzero`), onto which the tool-specific check categories (e.g. Polyspace‘s ZDV) can be mapped. The check result (safe, unsafe or undecided) is given in the check element’s *status* attribute. The tool may provide further details, e.g. reasons for an undecided status, in the *statusSupplement* attribute. Furthermore, the tool‘s internal check category and status are provided using the *internalCategory* rsp. *internalStatus* attributes. Finally, the analysis tool may provide free-form information about the result in the *annotation* field.

```
<<Check result collection>> =
  <ascccf:Checks>
    <ResultsFor executionReport="Report for ExampleProject-on-PPC
analysis">
      <Check id="1">
        <CCheck category="integerarithmetic.divbyzero" status="unsafe"
                internalCategory="divbyzero" internalStatus="unsafe"
annotation="">
          <LocID>1</LocID>
        </CCheck>
      </Check>
      <Check id="2">
        <CCheck category="assertion.user" status="safe"
statusSupplement="locally"
                internalCategory="user.assertion" internalStatus="safe"
                annotation="locally">
          <LocID>12</LocID>
        </CCheck>
      </Check>
      <Check id="20">
        <CCheck category="assertion.user" status="undecided"
                statusSupplement="function-body-missing"
                internalCategory="user.assertion" internalStatus="safe"
                annotation="function body missing">
          <LocID>22</LocID>
        </CCheck>
      </Check>
    </ResultsFor>
  </ascccf:Checks>
```

A.2.6. Failure Trace Collection

Finally, the failure trace collection provides additional information about the failed checks. However, this section remains to be designed.

```
<<Failure trace collection>> =
  <ascccf:FailureTraces>
    <TODO_ConcreteFailureTraces/>
  </ascccf:FailureTraces>
```

A.3. ExampleConfiguration.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<asef:Configuration xmlns:asef="http://todo.example.com/asef"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://todo.example.com/asef ASC3F.xsd">
  <asef:GlobalConfiguration>
    <asef:CommonConfiguration>

      <Meta configurationName="Example configuration"
          version="1.1"
          description="This configuration is an example.">
        <Maintainer>Felix Kutzner (KIT)</Maintainer>
      </Meta>

      <HardwareTargets>
        <HardwareTarget xsi:type="asef:HomogenousHardwareTarget" name="PPC32"
          endianness="big"
          unalignedDereferenceSupported="true"
          pointerSize="32"
          functionPointerSize="32"/>
        <HardwareTarget xsi:type="asef:HomogenousHardwareTarget"
          name="x86-64"
          endianness="small"
          unalignedDereferenceSupported="true"
          pointerSize="64"
          functionPointerSize="64"/>
      </HardwareTargets>

      <LanguageTargets>
        <LanguageTarget xsi:type="asef:CLanguageTarget" name="C99"
          signedEnums="true"
          signedBitfields="false"
          fpRoundingMode="downward"
          fpConstantRoundingMode="toNearest"
          enumType="int"
          inlineAssemblyHandlingMode="ignore"
          initializeStaticVariables="true"
          enableVolatile="true"
          standardRevision="C99">
            <Types>
              <Type name="char" size="8" alignment="8" atomic="false"/>
              <Type name="short" size="16" alignment="16" atomic="false"/>
              <Type name="int" size="32" alignment="32" atomic="false"/>
              <Type name="long" size="32" alignment="32" atomic="false"/>
              <Type name="long long" size="64" alignment="32" atomic="false"/>
              <Type name="float" size="32" alignment="32" atomic="false"/>
              <Type name="double" size="64" alignment="64" atomic="false"/>
            </Types>
          </LanguageTarget>

          <LanguageTarget xsi:type="asef:CLanguageSubtarget"
            name="C99 with includes"
            superTarget="C99">
            <PreprocessorDefinitions overrideParentDefinitions="false">
              <Definition identifier="STATIC_ANALYSIS"/>
            </PreprocessorDefinitions>
            <IncludeDirectories insertionMode="append">
              <DirectoryURI>$_LIBINCLUDES_$</DirectoryURI>
            </IncludeDirectories>
            <IncludeFiles insertionMode="append">
              <IncludeFile path="$_SYSINCLUDES_$/compiler_sys.h" local="true"/>
            </IncludeFiles>
          </LanguageTarget>

          <LanguageTarget xsi:type="asef:CLanguageSubtarget"
            name="LT for ExampleModule"
            superTarget="C99 with includes">
            <IncludeDirectories insertionMode="append">
              <DirectoryURI>$_EXAMPLEMODULEPATH_$/include</DirectoryURI>
            </IncludeDirectories>
          </LanguageTarget>
      </LanguageTargets>


      <CheckTargets>
        <CheckTarget xsi:type="asef:CCheckTarget" name="BasicChecks">
          <CorrectnessCheckCategory name="numeric.divbyzero"
              failureHandlingMode="wraparound"/>
          <CorrectnessCheckCategory name="mem.ptr.deref" failureHandlingMode="stop"/>
          <!-- ... -->
        </CheckTarget>
      </CheckTargets>

      <ExecutionModelTargets>
        <ExecutionModelTarget xsi:type="asef:CSynchronousExecutionModelTarget"
                            name="ExampleExecModel">
          <EntryPoints>
            <EntryPoint>main</EntryPoint>
          </EntryPoints>
        </ExecutionModelTarget>
      </ExecutionModelTargets>

      <SourceModules>
```

```
        <SourceModule name="ExampleModule" rootUri="$_EXAMPLEMODULESRC_$">
          <SourceFiles>
            <SourceFile uri="main.c" id="1"/>
            <SourceFile uri="dodgycode.c" id="2">
              <LanguageTargetExtension>
                <CLanguageSubtarget superTarget="auto">
                  <PreprocessorDefinitions overrideParentDefinitions="false">
                    <Definition identifier="ADD_MACRO" expansion="x+y"/>
                    <Definition identifier="KBD_PORT_ADDR" expansion="0x60"/>
                    <Definition identifier="KBD_STAT_ADDR" expansion="0x64"/>
                    <Definition identifier="KBD_CMD_ADDR" expansion="0x64"/>
                  </PreprocessorDefinitions>
                  <IncludeFiles insertionMode="append">
                    <IncludeFile path="HideProprietaryCExtensions.h" local="true"/>
                  </IncludeFiles>
                </CLanguageSubtarget>
              </LanguageTargetExtension>
            </SourceFile>
            <SourceFile uri="shadycode.c" id="3"/>
          </SourceFiles>

          <RequiredCStubs>
            <VisibilityControllingSymbols>
              <CVisibilityControllingSymbol name="ENABLE_LLBMC_STUBS"/>
              <CVisibilityControllingSymbol name="ENABLE_ASTREE_STUBS"/>
            </VisibilityControllingSymbols>
            <CUniversalStub uri="cstub://globalscope/open" group="posix_io"/>
            <CUniversalStub uri="cstub://globalscope/read" group="posix_io"/>
            <CUniversalStub uri="cstub://globalscope/write" group="posix_io"/>
            <CVisibilityControlledStub
uri="cstub://filescope/ExampleModule/2/read_from_kbd"/>
            <CAutogenStub uri="cstub://globalscope/malloc"/>
            <CAutogenStub uri="cstub://globalscope/free"/>
          </RequiredCStubs>
        </SourceModule>


        <SourceModule name="ExampleModule_stubs" rootUri="$_EXAMPLEMODULESTUBS_$">
          <SourceFiles>
            <SourceFile uri="groups/posix_io.c" id="1">
              <ImplementsStubs>
                cstub://globalscope/open
                cstub://globalscope/read
                cstub://globalscope/write
              </ImplementsStubs>
            </SourceFile>

            <SourceFile uri="fancy_rng.c" id="2">
              <ImplementsStubs>cstub://globalscope/fancy_rng</ImplementsStubs>
            </SourceFile>

            <SourceFile uri="filescope/ExampleModule/2/read_from_device.c" id="3">
              <ImplementsStubs>cstub://filescope/ExampleModule/2/read_from_kbd
              </ImplementsStubs>
            </SourceFile>
          </SourceFiles>
        </SourceModule>
        <SourceModule name="ExampleModule_joined">
          <SourceFiles>
            <SourceFile uri="someOtherFancyRNG.c" id="1"/>
          </SourceFiles>
          <RequiresModules>
            <RequiresModule name="ExampleModule"/>
            <RequiresModule name="ExampleModule_stubs">
              <ExcludingFilesProvidingStub uri="cstub://globalscope/fancy_rng"/>
            </RequiresModule>
          </RequiresModules>
        </SourceModule>
      </SourceModules>

      <AnalysisTasks>
        <AnalysisTask name="analyzeExampleModuleOnPPC"
                      missingRequiredCapabilityHandlingMode="abort">
          <ReportGeneratorConfiguration documentFormat="assume"/>
          <HardwareTarget>PPC32</HardwareTarget>
          <SourceModule>ExampleModule_joined</SourceModule>
          <LanguageTarget>LT for ExampleModule</LanguageTarget>
          <CheckTarget>BasicChecks</CheckTarget>
          <ExecutionModelTarget>ExampleExecModel</ExecutionModelTarget>
        </AnalysisTask>
        <AnalysisTask name="analyzeExampleModuleOnX86_64"
                      missingRequiredCapabilityHandlingMode="abort">
          <ReportGeneratorConfiguration documentFormat="assume"/>
          <HardwareTarget>x86-64</HardwareTarget>
          <SourceModule>ExampleModule_joined</SourceModule>
          <LanguageTarget>LT for ExampleModule</LanguageTarget>
          <CheckTarget>BasicChecks</CheckTarget>
          <ExecutionModelTarget>ExampleExecModel</ExecutionModelTarget>
        </AnalysisTask>
      </AnalysisTasks>
    </asef:CommonConfiguration>


    <asef:ToolConfigurations>
      <ToolConfiguration name="LLBMC" customParameters="-x">
        <ForAnalysisTasks>
          <AnalysisTaskName>PPC32</AnalysisTaskName>
          <AnalysisTaskName>x86_64</AnalysisTaskName>
        </ForAnalysisTasks>

        <LanguageTargetExtension>
          <CLanguageSubtarget superTarget="auto">
            <PreprocessorDefinitions overrideParentDefinitions="false">
              <Definition identifier="STATIC_ANALYSIS"/>
              <Definition identifier="ENABLE_LLBMC_STUBS"/>
            </PreprocessorDefinitions>
          </CLanguageSubtarget>
        </LanguageTargetExtension>

        <ToolSpecificConfiguration>
          <LoopBound>40</LoopBound>
        </ToolSpecificConfiguration>

      </ToolConfiguration>
    </asef:ToolConfigurations>
```

```
  </asef:GlobalConfiguration>

  <asef:LocalConfiguration>
    <URISubstitutionRules>
      <URISubstitutionRule token="$_LIBINCLUDES_$"
              substitution="file:///usr/include/libc"/>
      <URISubstitutionRule token="$_SYSINCLUDES_$" substitution="file:///usr/include"/>
      <URISubstitutionRule token="$_EXAMPLEMODULEPATH_$"
              substitution="file:///users/felix/projects/ExampleProject"/>
      <URISubstitutionRule token="$_EXAMPLEMODULESRC_$"
              substitution="$_EXAMPLEMODULEPATH_$/src"/>
      <URISubstitutionRule token="$_EXAMPLEMODULESTUBS_$"
              substitution="$_EXAMPLEMODULEPATH_$/verification/generatedStubs"/>
      <URISubstitutionRule token="$_EXAMPLEMODULEJOINED_$"
              substitution="$_EXAMPLEMODULEPATH_$/verification/joined"/>
    </URISubstitutionRules>
  </asef:LocalConfiguration>
</asef:Configuration>
```

A.4. ExampleReport.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!--XML-Beispieldatei von XMLSpy generiert v2016 (x64) (http://www.altova.com)-->
<asef:Report xmlns:asef="http://todo.example.com/asef"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  name="ExampleProjectAnalysis" sourceVersion="3.1"
  xsi:schemaLocation="http://todo.example.com/asef ASCCRF.xsd">
  <asef:Configuration>
    <asef:GlobalConfiguration>
      <asef:CommonConfiguration>
        <Meta configurationName="Example report" version="1.1"
              description="This configuration is an example.">
          <Maintainer>Felix Kutzner (KIT)</Maintainer>
        </Meta>
        <HardwareTargets>
          <HardwareTarget xsi:type="asef:HomogenousHardwareTarget" name="PPC32"
              endianness="big" unalignedDereferenceSupported="true" pointerSize="32"
              functionPointerSize="32"/>
          <HardwareTarget xsi:type="asef:HomogenousHardwareTarget" name="x86-64"
              endianness="small" unalignedDereferenceSupported="true" pointerSize="64"
              functionPointerSize="64"/>
        </HardwareTargets>
        <LanguageTargets>
          <LanguageTarget xsi:type="asef:CLanguageTarget" name="C99" signedEnums="true"
              signedBitfields="false" fpRoundingMode="downward"
              fpConstantRoundingMode="toNearest" enumType="int"
              inlineAssemblyHandlingMode="ignore" initializeStaticVariables="true"
              enableVolatile="true" standardRevision="C99">
            <Types>
              <Type name="char" size="8" alignment="8" atomic="false"/>
              <Type name="short" size="16" alignment="16" atomic="false"/>
              <Type name="int" size="32" alignment="32" atomic="false"/>
              <Type name="long" size="32" alignment="32" atomic="false"/>
              <Type name="long long" size="64" alignment="32" atomic="false"/>
              <Type name="float" size="32" alignment="32" atomic="false"/>
              <Type name="double" size="64" alignment="64" atomic="false"/>
            </Types>
          </LanguageTarget>
          <LanguageTarget xsi:type="asef:CLanguageSubtarget" name="C99 with includes"
              superTarget="C99">
            <PreprocessorDefinitions overrideParentDefinitions="false">
              <Definition identifier="STATIC_ANALYSIS"/>
            </PreprocessorDefinitions>
            <IncludeDirectories insertionMode="append">
              <DirectoryURI>$_LIBINCLUDES_$</DirectoryURI>
            </IncludeDirectories>
            <IncludeFiles insertionMode="append">
              <IncludeFile path="$_SYSINCLUDES_$/compiler_sys.h" local="true"/>
            </IncludeFiles>
          </LanguageTarget>
          <LanguageTarget xsi:type="asef:CLanguageSubtarget" name="LT for ExampleModule"
              superTarget="C99 with includes">
            <IncludeDirectories insertionMode="append">
              <DirectoryURI>$_EXAMPLEMODULEPATH_$/include</DirectoryURI>
            </IncludeDirectories>
          </LanguageTarget>
        </LanguageTargets>
        <CheckTargets>
          <CheckTarget xsi:type="asef:CCheckTarget" name="BasicChecks">
            <!-- TODO -->
          </CheckTarget>
        </CheckTargets>
        <ExecutionModelTargets>
          <ExecutionModelTarget xsi:type="asef:CSynchronousExecutionModelTarget"
              name="ExampleExecModel">
            <EntryPoints>
              <EntryPoint>main</EntryPoint>
            </EntryPoints>
          </ExecutionModelTarget>
        </ExecutionModelTargets>
        <SourceModules>
          <SourceModule name="ExampleModule" rootUri="$_EXAMPLEMODULESRC_$">
            <SourceFiles>
              <SourceFile uri="main.c" id="1"/>
              <SourceFile uri="dodgycode.c" id="2">
                <LanguageTargetExtension>
                  <CLanguageSubtarget superTarget="auto">
                    <PreprocessorDefinitions overrideParentDefinitions="false">
                      <Definition identifier="ADD_MACRO" expansion="x+y"/>
                      <Definition identifier="KBD_PORT_ADDR" expansion="0x60"/>
                      <Definition identifier="KBD_STAT_ADDR" expansion="0x64"/>
                      <Definition identifier="KBD_CMD_ADDR" expansion="0x64"/>
                    </PreprocessorDefinitions>
                    <IncludeFiles insertionMode="append">
                      <IncludeFile path="HideProprietaryCExtensions.h" local="true"/>
                    </IncludeFiles>
                  </CLanguageSubtarget>
                </LanguageTargetExtension>
              </SourceFile>
              <SourceFile uri="shadycode.c" id="3"/>
            </SourceFiles>
            <RequiredCStubs>
              <VisibilityControllingSymbols>
                <CVisibilityControllingSymbol name="ENABLE_LLBMC_STUBS"/>
                <CVisibilityControllingSymbol name="ENABLE_ASTREE_STUBS"/>
              </VisibilityControllingSymbols>
              <CUniversalStub uri="cstub://globalscope/open" group="posix_io"/>
              <CUniversalStub uri="cstub://globalscope/read" group="posix_io"/>
              <CUniversalStub uri="cstub://globalscope/write" group="posix_io"/>
              <CVisibilityControlledStub
                     uri="cstub://filescope/ExampleModule/2/read_from_kbd"/>
              <CAutogenStub uri="cstub://globalscope/malloc"/>
              <CAutogenStub uri="cstub://globalscope/free"/>
            </RequiredCStubs>
          </SourceModule>
          <SourceModule name="ExampleModule_stubs" rootUri="$_EXAMPLEMODULESTUBS_$">
            <SourceFiles>
              <SourceFile uri="groups/posix_io.c" id="1">
                <ImplementsStubs>
                cstub://globalscope/open
                cstub://globalscope/read
                cstub://globalscope/write
              </ImplementsStubs>
              </SourceFile>
              <SourceFile uri="fancy_rng.c" id="2">
                <ImplementsStubs>cstub://globalscope/fancy_rng</ImplementsStubs>
              </SourceFile>
              <SourceFile uri="filescope/ExampleModule/2/read_from_device.c" id="3">
                <ImplementsStubs>cstub://filescope/ExampleModule/2/read_from_kbd
                </ImplementsStubs>
              </SourceFile>
            </SourceFiles>
          </SourceModule>
          <SourceModule name="ExampleModule_joined">
            <SourceFiles>
              <SourceFile uri="someOtherFancyRNG.c" id="1"/>
            </SourceFiles>
            <RequiresModules>
              <RequiresModule name="ExampleModule"/>
              <RequiresModule name="ExampleModule_stubs">
                <ExcludingFilesProvidingStub uri="cstub://globalscope/fancy_rng"/>
              </RequiresModule>
            </RequiresModules>
          </SourceModule>
        </SourceModules>
        <AnalysisTasks>
          <AnalysisTask name="analyzeExampleModuleOnPPC"
              missingRequiredCapabilityHandlingMode="abort">
            <ReportGeneratorConfiguration documentFormat="assume"/>
            <HardwareTarget>PPC32</HardwareTarget>
            <SourceModule>ExampleModule_joined</SourceModule>
            <LanguageTarget>LT for ExampleModule</LanguageTarget>
            <CheckTarget>BasicChecks</CheckTarget>
            <ExecutionModelTarget>ExampleExecModel</ExecutionModelTarget>
          </AnalysisTask>
          <AnalysisTask name="analyzeExampleModuleOnX86_64"
              missingRequiredCapabilityHandlingMode="abort">
            <ReportGeneratorConfiguration documentFormat="assume"/>
            <HardwareTarget>x86-64</HardwareTarget>
            <SourceModule>ExampleModule_joined</SourceModule>
            <LanguageTarget>LT for ExampleModule</LanguageTarget>
            <CheckTarget>BasicChecks</CheckTarget>
            <ExecutionModelTarget>ExampleExecModel</ExecutionModelTarget>
```

```
          </AnalysisTask>
        </AnalysisTasks>
      </asef:CommonConfiguration>
      <asef:ToolConfigurations>
        <ToolConfiguration name="LLBMC" customParameters="-x">
          <ForAnalysisTasks>
            <AnalysisTaskName>PPC32</AnalysisTaskName>
            <AnalysisTaskName>x86_64</AnalysisTaskName>
          </ForAnalysisTasks>
          <LanguageTargetExtension>
            <CLanguageSubtarget superTarget="auto">
              <PreprocessorDefinitions overrideParentDefinitions="false">
                <Definition identifier="STATIC_ANALYSIS"/>
                <Definition identifier="ENABLE_LLBMC_STUBS"/>
              </PreprocessorDefinitions>
            </CLanguageSubtarget>
          </LanguageTargetExtension>
          <ToolSpecificConfiguration>
            <LoopBound>40</LoopBound>
          </ToolSpecificConfiguration>
        </ToolConfiguration>
      </asef:ToolConfigurations>
    </asef:GlobalConfiguration>
    <asef:LocalConfiguration>
      <URISubstitutionRules>
        <URISubstitutionRule token="$_LIBINCLUDES_$"
              substitution="file:///usr/include/libc"/>
        <URISubstitutionRule token="$_SYSINCLUDES_$" substitution="file:///usr/include"/>
        <URISubstitutionRule token="$_EXAMPLEMODULEPATH_$"
              substitution="file:///users/felix/projects/ExampleProject"/>
        <URISubstitutionRule token="$_EXAMPLEMODULESRC_$"
              substitution="$_EXAMPLEMODULEPATH_$/src"/>
        <URISubstitutionRule token="$_EXAMPLEMODULESTUBS_$"
              substitution="$_EXAMPLEMODULEPATH_$/verification/generatedStubs"/>
        <URISubstitutionRule token="$_EXAMPLEMODULEJOINED_$"
              substitution="$_EXAMPLEMODULEPATH_$/verification/joined"/>
      </URISubstitutionRules>
    </asef:LocalConfiguration>
  </asef:Configuration>

  <asef:ExecutionReports>
    <asef:ExecutionReport name="Report for ExampleProject-on-PPC analysis"
      analysisTask="analyzeExampleModuleOnPPC"
      toolParameters="LLBMC"
      analysisBeginDate="2016-12-17T09:30:47Z"
      analysisFinishDate="2016-12-17T09:30:49Z"
      analysisComputerName="i11pc164"
      toolName="LLBMC">
      <SourceCodeProcessingMessages>
        <SourceCodeProcessingMessage xsi:type="asef:FreeformSourceCodeProcessingMessage"
              locationID="100" msg="warning: '&amp;&amp;' within '||'
                     [-Wlogical-op-parentheses]"/>
      </SourceCodeProcessingMessages>
      <CheckerMessages>
        <!-- ... -->
      </CheckerMessages>
      <ConfigDeviations>
        <!-- ... -->
      </ConfigDeviations>
    </asef:ExecutionReport>
  </asef:ExecutionReports>

  <asef:SourceStorages>
    <Storage xsi:type="asef:CSourceFile" id="1" path="$_EXAMPLEMODULESRC_$/main.c"
       hashSum="1b826051506f463f07307598fcf12fd6" preprocessed="false"
       originModule="ExampleModule" idInOriginModule="1"/>
    <Storage xsi:type="asef:CSourceFile" id="2" path="$_EXAMPLEMODULESRC_$/dodgycode.c"
       hashSum="3b5337aa426bb547efefb97edec54e3e" preprocessed="false"
       originModule="ExampleModule" idInOriginModule="2"/>
    <Storage xsi:type="asef:CSourceFile" id="3" path="$_EXAMPLEMODULESRC_$/shadycode.c"
       hashSum="ff702f10bebfa2f1508deb475ded2d65" preprocessed="false"
       originModule="ExampleModule" idInOriginModule="3"/>
    <Storage xsi:type="asef:CHeaderFile" id="4"
       path="$_EXAMPLEMODULEPATH_$/include/ExampleModule.h"
       hashSum="2f702f10bebfa2f1508deb475ded2d65" preprocessed="false"
       includeDirectory=""/>
    <Storage xsi:type="asef:CHeaderFile" id="5" path="compiler_sys.h"
       hashSum="3f702f10bebfa2f1508deb475ded2d65" preprocessed="false"
       includeDirectory="$_SYSINCLUDES_$"/>
    <Storage xsi:type="asef:CHeaderFile" id="6" path="HideProprietaryCExtensions.h"
       hashSum="4f702f10bebfa2f1508deb475ded2d65" preprocessed="false"
       includeDirectory="$_EXAMPLEMODULESRC_$"/>
    <Storage xsi:type="asef:CSourceFile" id="10"
       path="$_EXAMPLEMODULESTUBS_$/groups/posix_io.c"
       hashSum="8633b81a334995b50b53df83581af093" preprocessed="false"
       originModule="ExampleModule_stubs" idInOriginModule="1"/>
    <Storage xsi:type="asef:CSourceFile" id="11" path="$_EXAMPLEMODULESTUBS_$/fancy_rng.c"
       hashSum="b06f74ff6378f4a2629621b3d8aa935f" preprocessed="false"
       originModule="ExampleModule_stubs" idInOriginModule="2"/>
    <Storage xsi:type="asef:CSourceFile" id="12"
       path="$_EXAMPLEMODULESTUBS_$/filescope/ExampleModule/4/read_from_device.c"
       hashSum="c143a9ae806ab2c93ad4f4f593173bf0" preprocessed="false"
       originModule="ExampleModule_stubs" idInOriginModule="3"/>
    <Storage xsi:type="asef:CSourceFile" id="20"
       path="$_EXAMPLEMODULEJOINED_$/someOtherFancyRNG.c"
       hashSum="591d99a6a84b1e1dbb44395a3fa27d64" preprocessed="false"
       originModule="ExampleModule_joined" idInOriginModule="1"/>
    <Storage xsi:type="asef:CFilelessPreprocessorDefinition" id="30" >
      <Definition identifier="ADD_MACRO" expansion="x+y"/>
    </Storage>
  </asef:SourceStorages>

  <asef:Locations>
    <Location xsi:type="asef:CSourceRealLocation" id="1" storageID="2"
       lineNo="22" colNo="8"/>
    <Location xsi:type="asef:CSourceRealLocation" id="2" storageID="30"
       lineNo="1" colNo="2"/>
    <Location xsi:type="asef:CSourceRealLocation" id="10" storageID="3"
       lineNo="1" colNo="1"/>
    <Location xsi:type="asef:CSourceRealLocation" id="11" storageID="3"
       lineNo="1" colNo="1"/>
    <Location xsi:type="asef:CSourceMacroLocation" id="12" spellingLocID="10"
       expansionLocID="11"/>
    <Location xsi:type="asef:CSourceRealLocation" id="20" storageID="2"
       lineNo="22" colNo="12"/>
    <Location xsi:type="asef:CSourceRealLocation" id="21" storageID="2"
```

```
       lineNo="22" colNo="20"/>
    <Location xsi:type="asef:RangeLocation" id="22" beginLocID="20" endLocID="21"/>
    <Location xsi:type="asef:PlaintextRealLocation" id="30" storageID="20"
       lineNo="5" colNo="53"/>
    <Location xsi:type="asef:PlaintextLineLocation" id="31" storageID="20"
       lineNo="5"/>
  </asef:Locations>

  <asef:Checks>
    <ResultsFor executionReport="Report for ExampleProject-on-PPC analysis">
      <Check xsi:type="asef:CCheck" id="1" category="integerarithmetic.divbyzero"
              status="unsafe" internalCategory="divbyzero" internalStatus="unsafe"
              annotation="">
        <LocID>1</LocID>
      </Check>
      <Check xsi:type="asef:CCheck" id="2" category="assertion.user" status="safe"
              statusSupplement="locally" internalCategory="user.assertion"
              internalStatus="safe" annotation="locally">
        <LocID>12</LocID>
      </Check>
      <Check xsi:type="asef:CCheck" id="20" category="assertion.user" status="undecided"
              statusSupplement="function-body-missing" internalCategory="user.assertion"
              internalStatus="safe" annotation="function body missing">
        <LocID>22</LocID>
      </Check>
    </ResultsFor>
  </asef:Checks>

  <asef:FailureTraces>
  </asef:FailureTraces>
</asef:Report>
```