Collaborative Large-scale Integrating Project

OPENCOSS
Open Platform for EvolutioNary Certification Of Safety-critical Systems

Integration with development and safety assurance tools
D6.4

<table>
<thead>
<tr>
<th>Work Package:</th>
<th>WP6: Evolutionary Evidential Chain</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dissemination level:</td>
<td>PU</td>
</tr>
<tr>
<td>Status:</td>
<td>Final</td>
</tr>
<tr>
<td>Date:</td>
<td>10 June 2013</td>
</tr>
<tr>
<td>Responsible partner:</td>
<td>Eric Verhulst (Altreonic)</td>
</tr>
<tr>
<td>Contact information:</td>
<td><a href="mailto:eric.verhulst@altreonic.com">eric.verhulst@altreonic.com</a></td>
</tr>
</tbody>
</table>

PROPRIETARY RIGHTS STATEMENT
This document contains information proprietary to the OPENCOSS Consortium. Neither this document nor the information contained herein shall be used, duplicated or communicated by any means to any third party, in whole or in parts, except with prior written consent of the OPENCOSS consortium.
# Contributors

<table>
<thead>
<tr>
<th>Names</th>
<th>Organisation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Eric Verhulst, Bernhard Sputh, Leonel Braga</td>
<td>Altreonic</td>
</tr>
<tr>
<td>Dariusz Oszczędowski</td>
<td>PSF</td>
</tr>
<tr>
<td>Thierry VENTADOUR, Hugues DARRIET</td>
<td>INS (previously DNV)</td>
</tr>
</tbody>
</table>

## Document History

<table>
<thead>
<tr>
<th>Version</th>
<th>Date</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>V.0.1</td>
<td>2013-04-04</td>
<td>Document creation and initial ToC</td>
</tr>
<tr>
<td>V.0.2</td>
<td>2013-04-15</td>
<td>Update of 1.2 Relationship with other deliverables (INS)</td>
</tr>
<tr>
<td>V.0.3</td>
<td>2013-04-26</td>
<td>PSF evidence types description and tools integration ideas</td>
</tr>
<tr>
<td>V.0.4</td>
<td>2013-04-29</td>
<td>ALT on formal models as evidence</td>
</tr>
<tr>
<td>v.0.5</td>
<td>2013-05-11</td>
<td>ALT adding figures</td>
</tr>
<tr>
<td>v.0.6</td>
<td>2013-05-21</td>
<td>Updated after internal review</td>
</tr>
<tr>
<td>v.0.7</td>
<td>2013-05-27</td>
<td>Updated with OSLC based section after additional comments</td>
</tr>
<tr>
<td>V.0.8</td>
<td>2013-06-04</td>
<td>Consolidated all comments. Ready for PB.</td>
</tr>
<tr>
<td>v.0.9</td>
<td>06-06-2013</td>
<td>Update after PB comments</td>
</tr>
<tr>
<td>V.0.95</td>
<td>07-06-2013</td>
<td>IKV tool description and evidence types</td>
</tr>
<tr>
<td>V.1.0</td>
<td>10-06-2013</td>
<td>Final version for EC</td>
</tr>
</tbody>
</table>
TABLE OF CONTENTS

List of Figures.................................................................5
Abbreviations .............................................................5

1 Executive Summary .................................................. 6

2 Introduction ...................................................................... 7
  2.1 Context in the OPENCOSS project .............................. 7
  2.2 Relationship with other Deliverables ........................... 8
  2.3 Structure of the Document ........................................... 8

3 Analysis of the feasibility of integration with existing tools .............................................. 8
  3.1 Evidence in GoedelWorks (ALT) ..................................... 9
    3.1.1 General certification approach in GoedelWorks .......... 9
    3.1.2 Concrete application in the context of using formal techniques for software development ................. 11
      3.1.2.1 Formal models as a development aid and for the verification of software 11
      3.1.2.2 Example: the OpenComRTOS Event Hub ......................... 12
        3.1.2.2.1 Summary description of OpenComRTOS ........ 12
        3.1.2.2.2 The Event-Hub .................................................. 13
        3.1.2.2.3 Dependency trees in GoedelWorks .......................... 13
    3.1.3 Data exchange with the OPENCOSS platform ............ 18
      3.1.3.1 Data exchange possibilities with the OPENCOSS platform ...... 18

  3.2 Evidence in the Parasoft products portfolio (PSF) ......................................................... 20
    3.2.1 Brief overview of the Parasoft products ................. 20
    3.2.2 Evidence types suitable for the OPENCOSS platform ........ 21
      3.2.2.1 Static code analysis results ........................................... 21
      3.2.2.2 Unit tests results ....................................................... 22
      3.2.2.3 Code review results .................................................... 23
      3.2.2.4 Manual tests results ................................................... 23
      3.2.2.5 Tasks/defects/requirements state ................................. 24

  3.3 Evidence in medini analyze (IKV) .................................... 24
    3.3.1 General overview on the tool ................................. 24
    3.3.2 Supported evidence types ......................................... 26
      3.3.2.1 HARA ................................................................. 26
      3.3.2.2 FMEA/FTA ......................................................... 26
      3.3.2.3 Goals and Requirements ........................................ 26
      3.3.2.4 Architecture ......................................................... 26
      3.3.2.5 Traces ............................................................... 26
      3.3.2.6 Supporting artefacts .............................................. 26
    3.3.3 Potential data exchange within OPENCOSS ............... 26

4 Initial ideas for an integration API as seen by ParaSoft .................................................... 27
  4.1 Establishing a common registry of evidence types provided by external tools .................. 27
  4.2 Establishing a common format for evidence import/export from external tools ............... 29
  4.3 Data integrity assurance .................................................. 29
  4.4 Unique identifiers for every imported evidence ............................................................. 29
  4.5 Pull + push import/export .................................................. 29
4.6 Relation between imported/remote evidence traced by unique identifiers.........................29

5 Overview of OSLC as data exchange protocol.................................................................30
5.1 Short introduction on IFEST ....................................................................................30
5.2 Short introduction on OSLC....................................................................................30
5.3 Guidelines from the IFEST project and the OSLC community.................................30
5.4 Consequences of selecting OSLC...........................................................................32

6 Conclusions..................................................................................................................33

7 References.....................................................................................................................34

8 Appendix A. Selected tools used by OPENCOSS partners ..............................................35
8.1 Tools used by project partners................................................................................35
8.2 Other tools developed/commercialized by other partners ........................................35
8.3 Other possible tools .................................................................................................36
List of Figures
Figure 1 Functional decomposition for the OPENCOSS tool platform .................................................. 7
Figure 2 GoedelWorks meta-meta-model .............................................................................................. 9
Figure 3 Subset of Hub-Event process trees .......................................................................................... 11
Figure 4 Dependency tree for a top-level Requirement ......................................................................... 15
Figure 5 Project Planning Dependency Tree ........................................................................................ 17
Figure 6 Parasoft product family ........................................................................................................ 20
Figure 7 Evidence taxonomy .............................................................................................................. 28

Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARRL</td>
<td>Assured Reliability and Resilience Level</td>
</tr>
<tr>
<td>CCL</td>
<td>Common Certification Language</td>
</tr>
<tr>
<td>CENELEC</td>
<td>Comité Européen de Normalisation Electrotechnique (European Committee for Electrotechnical Standardization)</td>
</tr>
<tr>
<td>DoW</td>
<td>OPENCOSS Description of Work</td>
</tr>
<tr>
<td>DX.Y</td>
<td>OPENCOSS deliverable X.Y</td>
</tr>
<tr>
<td>IEEE</td>
<td>Institute of Electrical and Electronics Engineers</td>
</tr>
<tr>
<td>RTOS</td>
<td>Real-Time Operating System</td>
</tr>
<tr>
<td>TX.Y</td>
<td>OPENCOSS task X.Y</td>
</tr>
<tr>
<td>V&amp;V</td>
<td>Verification and Validation</td>
</tr>
<tr>
<td>WP</td>
<td>OPENCOSS work package</td>
</tr>
<tr>
<td>ALM</td>
<td>Application Lifecycle Management</td>
</tr>
<tr>
<td>BTS</td>
<td>Bug Tracking System</td>
</tr>
<tr>
<td>SCM</td>
<td>Source Code Management</td>
</tr>
<tr>
<td>OSLC</td>
<td>Open Systems for Lifecycle Collaboration</td>
</tr>
<tr>
<td>URI</td>
<td>Universal Resource Identifier</td>
</tr>
<tr>
<td>URL</td>
<td>Universal Resource Locator</td>
</tr>
<tr>
<td>RDF</td>
<td>Resource Description Framework</td>
</tr>
<tr>
<td>HTTP</td>
<td>Hypertext Transfer Protocol</td>
</tr>
<tr>
<td>GUID</td>
<td>Globally Unique Identifier</td>
</tr>
<tr>
<td>UID</td>
<td>Unique Identifier</td>
</tr>
</tbody>
</table>
1 Executive Summary

WP6 will define part of the OPENCOSS safety certification management infrastructure together with WP7. While WP6 centers on the evidence chain as collected from different sources (including development and safety assurance tools), WP7 is more concerned with the processes view of certification (specification and execution).

WP6 has the following specific objectives:

- Analyze and assess the state of the art and state of the practice in terms of approaches for managing safety evidence.
- Develop detailed technical requirements for this WP following the high-level requirements defined in WP2.
- Identify success metrics for the evolutionary evidential chain service infrastructure.
- Design and implement a set of OPENCOSS platform services for safety certificates life cycle (creation, configuration, validation, etc.), integration of evidence items with development and safety assurance tools (requirement specification, design, code generation, safety analysis, testing, etc), integration and linking with the CCL (WP4).
- Guarantee the integration and interoperability of the OPENCOSS platform services with development and safety assurance tools providing evidence for certification.

This document (D6.4) is the third deliverable of WP6, and corresponds to the deliverable for T6.3.

This task aims at guaranteeing the interoperability of the OPENCOSS platform with tools used in the development and assurance of safety-critical systems, whereby evidence can be manually generated or automatically generated by the tools themselves (code generators, testing tools, safety analysis tools, etc.). The challenge is to be able to gather evidence from different types of tools by means of standardized and well-defined adapters or exchange tools.

During software development and safety assurance, different kinds of tools may be employed. For example, different forms of software testing could be used to validate and/or verify various parts of a system under development. Sections of code can be written in such a way that they can automatically be proven correct via an external theorem prover. A section of a program that can be logically or mathematically proven correct can be considered to be more trustworthy than a section that has only been tested.

The task will examine whether it is possible or not to define a set of interfaces and protocols to communicate with a selected set of safety-critical development tools. While the goal is not to cover all the possible tools, OPENCOSS will provide important adaptors or at least a template scheme with an emphasis on the tools commercialized by the project partners. This will be open and extensible for further integrations.

It is expected that the deliverable will be updated as the OPENCOSS CCL and platform matures and the requirements for the OPENCOSS tool platform evolve. Therefore, new versions of this deliverable will be released when needed.
2 Introduction

2.1 Context in the OPENCOSS project

The OPENCOSS project aims to devise a common certification framework that spans different vertical markets for railway, avionics, and automotive industries, and to establish an open-source safety certification infrastructure (hereafter referred to as the OPENCOSS tool platform). The infrastructure is being realised as a tightly-integrated solution, supporting interoperability with existing development and assurance tools. The ultimate goal of the project is to bring about substantial reductions in recurring safety certification costs, and at the same time to increase product safety through the introduction of more systematic certification practices. Both are expected to boost innovation and system upgrades considerably.

Among other areas, WP6 is concerned with interfacing or using data generated by development and safety assurance tools external to the OPENCOSS platforms (Figure 1).

Figure 1 Functional decomposition for the OPENCOSS tool platform
2.2 Relationship with other Deliverables

D6.4 is related to other OPENCOSS deliverables, which have either served as input, with which consistency must be kept, or that will use its results. These deliverables, and the relationships with them, are as follows:

- D6.2 “Detailed requirements for evidence management of the OPENCOSS platform”
- D6.3 “Specification of the evidence management service infrastructure” - provides D6.4 with functional and non-functional requirements related to evidence functional areas and services to be satisfied by the implemented OPENCOSS platform.
- D3.2 “Integration requirements and test plan” defines the unit testing related to the D6.4 specification of adapters to development and safety assurance tools.
- D4.1 “A baseline for the Common Certification Language” and subsequent CCL releases (D4.4). These WPs aim at defining a common conceptual and notational framework for specifying certification assets, as a means to discuss abstract notations from different domains. This framework is called CCL (Common Certification Language). The CCL should cover three conceptual levels:
  - description of safety cases,
  - description of evidence characterization,
  - description of compliance management.

2.3 Structure of the Document

In order to gain a better understanding of the problem statement, i.e. how external assurance tools can be interfaced with the OPENCOSS platform, an overview of available tools was created, listed in Annex A. This list consists of 53 tools identified or known by consortium partners. Given that interfacing with all these tools, whereby each tool must be understood in how it generates and presents data, would be prohibitive, two tools were selected for a deeper analysis in the first version of the deliverable.

The selected tools are GoedelWorks from Altreonic, a toolsuite from Parasoft and medini analyze from IKV. These were selected because they are well known by the participating contributors of this report and while representative the represent diverse examples of tools with some overlapping of tools used in the context of safety assurance.

The report is structured as follows:

- Project evidence generated in Altreonic’s GoedelWorks portal is examined in the context of developing the Event service in the OpenComRTOS project. In particular, attention is paid on how formal modelling complements the certification evidence.
- Next the evidence generated by Parasoft tools is presented and examples are given.
- Next the methods and evidence types handled by medini analyze are shortly presented.
- The last section examines how a data exchange could be implemented.
- In the annex a more complete list of safety assurance related tools is listed.

3 Analysis of the feasibility of integration with existing tools

Annex A provides a list of tools in use or mentioned by OPENCOSS partners that are candidates for interfacing with the OPENCOSS tool platform. Obviously, all the tools generate the data from a different perspective, and use internally different data formats and ways of storing the data. In addition,
Integration with development and safety assurance tools

Terminology, semantics, and relationships reflected in the tools differ. Assuming that a mapping (approximate or complete) is possible, several ways of interfacing between the OPENCOSS tool platform and the various tools are possible:

1. By import/export functions using an intermediate format (e.g. XML, CSV, ...). The latter can be based on the CCL.
2. By an API that allows to read and write the database in real-time.

Both methods still require human intervention to define the mappings between the data items of the tools and the data items of the OPENCOSS platform, or at least to verify them case by case. In particular the data must semantically map as well as their attributes as e.g. specified in the CCL.

In order to gain a better understanding of the integration challenges, two tools were selected. Firstly, it is analysed how they work, and secondly the generated data and artefacts are classified in view of a mapping to the OPENCOSS CCL.

3.1 Evidence in GoedelWorks (ALT)

3.1.1 General certification approach in GoedelWorks

The GoedelWorks project development environment is based on a generic systems engineering meta-meta-model (called “systems grammar”; Figure 2) that can be instantiated to a domain-specific or an
organisation-specific meta-model. A subsequent step is then to use it for a concrete project whereby the GoedelWorks portal (and its database) becomes a specific instance of a system developed, adhering to the GoedelWorks systems grammar. During project execution, users create, modify, complete and approve the project entities and create the dependency links between the project entities. The project entities are stored in a project specific database collection all project artefacts produced during the project’s execution. The terms used in this system grammar are capitalised when used in the text below to differentiate them from their natural language counterparts, even if the semantics partially overlap.

In GoedelWorks, three orthogonal but complementary views cover the systems engineering domain. A System is considered to be the result of the combination of these 3 views.

1. The Process view: defining how a system is developed. In particular this can be derived from e.g. safety standards.
2. The Architectural view: defining the system as a set of Interacting entities.
3. The Project view: this model defines the partial order of the steps, called a Flow, to be executed to go from initial Requirements to a concrete, eventually certifiable, System.

The core of each of these views is called a Work Package, which takes as input Specifications and Resource definitions and produces Work Products as a result. Work Products can either be Process-related, in case they generally correspond with the evidence artefacts required for e.g. certification (documents, reports, procedures; etc.), or are Architecture-related, in case they correspond with the Items that make up the System. Before the Kick-Off of a specific Project, the Process Work Products are not yet instantiated. Hence they are Resources for a Project Work Package. Examples are guidelines, templates, and procedures.

The partial order of the Project Flow defines dependency relationships. While these do not dictate a specific execution in time (allowing the modelling of both concurrent and iterative or agile types of engineering processes), they define the precedence relationships that must be followed in order to approve Work Products and hence certify a System.

Each Work Package follows a predefined pattern that defines the Activities and Artefacts from Kick-Off to Work Product release that must be generated in a managed, certifiable Project. In total, about 36 Activities and Artefacts are defined. They follow the generic pattern of: Planning - Development - Verification - Testing - Integration - Validation – Review, as one finds in most safety and engineering related standards. In the Architecture view, initially defined ARRL levels (Assured Reliability and Resilience Levels) [ARRL] are handled as Requirements and Specifications that need to be fulfilled by the Architecture and the system’s composing subsystems and externally acquired components. These ARRL level allow for composable safety by defining composition rules as a function of the required Safety Integrity Levels.

Hence, in GoedelWorks’ methodology assurance that a System meets certification criteria is based on a two-level approach:

1. The dependency chain (embodying a traceability Requirement) from initial- top level Requirements to the final System and its components as delivered only contains only Approved nodes in the graph.
2. Each Approval is justified by Arguments and Evidence, duly recorded and reviewed.

Figure 3 shows the top level meta-meta-model used in GoedelWorks for the specific case of the Event Hub in the OpenComRTOS project (for details see further):
3.1.2 Concrete application in the context of using formal techniques for software development

3.1.2.1 Formal models as a development aid and for the verification of software

In the GoedelWorks methodology, a Requirement is mapped onto a Specification by a refinement process. The transition is complete when the Specification can be tested and verified in an unambiguous way. Therefore, it is the input for a Testing Activity that includes the Test Plan, the actual Testing and the Testing report.

Before Testing can start (in the context of verifying the Specifications), the result of the Development needs to be Verified against the Process-related Specifications, so as to avoid any danger that Testing might fail due to an erroneous Development Process execution. This aims at reducing the residual errors that are due to development mistakes Testing is started. After Testing, the Test Report is produced and Reviewed, resulting in a Test Review Report that upon approval generates an evidence trace from Requirements till the Testing that confirms that the Item meets the Specifications.

When formal methods are involved, additional evidence is generated whereby formal models are developed and verified, i.e. invariants are specified and confirmed by model checking. This is the equivalent of Testing and results in a report that confirms the results of the model checking. This report is subsequently reviewed and results in an additional Review Report. Together with the other Review Reports, for example of reviews that confirm that the formal models are consistent with the system being
developed, it provides the evidence that the item meets the Specifications. The added value of the formal 
model-checking activities is that the confidence, or rather the Assurance level has been raised.

Both Review Reports are Evidence that contribute to the further confirmation activities leading to 
Certification. One must keep in mind that due to the methodology that each Report carries with it the 
Approval Trace of all Preceding Project Activities and Artefacts. This dependency trace is part of the 
Evidence.

A key point is that the Formal Models (and proven Invariants as Formal Specifications) must be 
homomorphic to the Implementation Models and its Specifications - hence, this is part of the evidence to 
be provided. This is less trivial than it looks at first sight. Firstly, Formal Models are inherently more abstract 
than Implementation Models. Low-level, implementation-level details are often not relevant whereas if 
taken into account they result in the state space to explode even faster and hence limiting the complexity 
of the models that can be formally checked.

Secondly, any implementation language has specific semantics and is tied in with an implementation 
processor and the selected compiler. Note, that other approaches are also possible. For example, source 
code can be generated from formally-proven models. While yielding very high confidence in the code, this 
technique is less flexible for evolving software and the current state of the art is weak when it comes to 
concurrent software, hence it is mainly used for sequential code that can be described with a state machine 
or state graph.

To understand this interaction between formal models and implementation, we examined a subset of the 
Activities and Evidence generated during the development of the OpenComRTOS RTOS. During this project, 
Formal Methods were used (mainly TLA+ with the TLC model checker) [TLA]. Prior to the development, 
Formal Models were developed and model checked to support the development of the architecture and 
the OS services, in particular the distributed semantics. After development, new Formal Models were 
developed derived from the source code to confirm the correctness of the implementation (Confirmation 
measure), as well as unit Test and typical stress Tests to gain more confidence in the developed software. 
The combined use of these methods raised the confidence and Assurance level in the released product.

3.1.2.2 Example: the OpenComRTOS Event Hub

In order to gain a better insight into how the GoedelWorks data can be interfaced with the OPENCOSS 
platform, it is not sufficient to reason at the data description level. Concrete data needs to be examined to 
see how it fits the CCL and what are the practical issues that can be encountered. As an example we took 
the Event service as documented in the OpenComRTOS project imported in a GoedelWorks portal.

3.1.2.2.1 Summary description of OpenComRTOS

OpenComRTOS is a formally developed network-centric RTOS [OCR1, OCR2]. It supports systems from one 
to many processors, independently of the type of processor, the type of communication mechanism, and 
the topology of the target system. In OpenComRTOS parlance, the target system is composed of “Nodes” 
connected by “Links” (even when the nodes are core in the same chip communicating over memory).

Each Node is managed by a “Kernel Task” providing Kernel Services and Priority-based pre-emptive 
scheduling to a number of Tasks. These Tasks can be application Tasks as well as Driver Tasks.

The Tasks synchronise and communicate across the target system using intermediate “Hub” entities. The 
Hubs are implementations of a Guarded Atomic Action whereby Tasks synchronise on a service-specific
Boolean condition (called the “synchronisation predicate”) followed by an action (called the “action predicate”). OpenComRTOS is written in C and mainly supports applications in C. The Tasks call the Hub services using C-function calls, but the kernel itself operates by using a Packet-switching architecture, allowing Tasks to use Hubs even when on different Nodes in the target system.

The Hubs emulate the common services found in most RTOS like Events: counting Semaphores, Fifos, Resource locks, and Memory Pools. They share a common Hub functionality, contributing to a significant reduction in code size. In this report, we consider only the Event Hub, in order to investigate the link between the formal models and the implementation models and how the data can be exchanged with the OPENCROSS platform. Traceability was also limited to development, test and verification activities. The simplified process is depicted in Figure 2. While the Event Hub is one of the simplest services, it still requires most of the RTOS kernel level functionality like scheduling and is representative for all services.

3.1.2.2 The Event-Hub

The simplest of all Hubs is the Event. Two Tasks can use it to synchronise on a binary condition, often associated with a hardware-related event, or when 2 Tasks need to synchronise exclusively with each other, implementing a so-called 1-to-1 synchronisation semantics. This can be seen as a restricted case of the counting Semaphore, whereby N-to-N synchronisation is allowed. In the case of the Event Hub, one Task will raise the Event and another task will wait on the Event (hence the implementation has corresponding waiting lists). Synchronisation only happens when the Boolean Event is true; otherwise, one of the Tasks will have to wait until such an event happens. This is the typical use case of blocking or waiting semantics. Other use cases are the non-blocking semantics (test and return) and the blocking with time-out semantics. Hence, in principle no events will ever be lost or over-written. The only exception is when an Interrupt Service Routine (ISR) raises and Event, as an ISR has no proper context and hence cannot wait or receive a return value from the service.

We consider here only the blocking semantics between Tasks synchronising the Event-Hub. The implementation was formally model checked using TLA+/TLC. The semantics of TLA+ reflect any state change everywhere in the model, hence there was no need to explicitly model the fact that Tasks and Hubs can be placed on different Nodes.

3.1.2.2.3 Dependency trees in GoedelWorks

In this section we outline the organisation of the artefacts according to the GoedelWorks grammar. From a top level Requirement (that is generic by definition), two classes of Specifications were derived.

- Implementation Specifications: the application-level functional Specifications. These are given in the format of the C-level API and a descriptive text as found in the programmer’s manual.
- Formal Specifications: the functional Specifications of the Formal TLA+ model, complemented by the Invariants that were proven by the TLC model checker.

The first class of Specifications are mapped onto the Implementation Model (C source code), whereas the Formal Specifications were mapped onto the Formal Models (TLA+ source code). Each file (C-source code or TLA+ source code) is called an Item. Functions that are called from a higher level function are considered as structural components of the item (whereas for the compilation phase these are considered as dependencies).

Several observations need to be made:
1. Each of the Models requires many supporting functions.
2. Specified functionality, in this case the Hub-Event, is spread over multiple implementation Items.
For example the kernel loop and scheduling support co-defines the behavioural properties.

3. This is true for the implementation c-source code and to a lesser extent also for the TLA+ models. The latter make it clearer that invariant properties are part of the Model whereas when looking at the source code, invariant properties are considered as implicit properties of an implementation.

Next, the Implementation Models are linked with the available Tests, unit Tests as well as stress Tests that exercise the code when integrated with the rest of the System (as they require building a small test application).

In Figure 4, the graphical dependency tree generated with a high-level Requirement (“Deterministic Kernel Services”) as root is displayed. The generated tree was filtered for Event services and limited to Development, Verification, and Review activities. Development is concurrent between the formal modelling activities and implementation activities. Both activities are Verified, Tests developed and executed executed on the implementation and formal models are checked using the corresponding formal model checkers. This highlights the difference between testing and formal model checking. While the first activity requires a test to be developed for each case, formal model checkers traverse the whole state space and continuously looks for traces leading to illegal states (e.g. specified as invariants). While testing is likely to be incomplete (as the state space is too large), formal models cannot not always take into account the full environment in which a software module is used although it will often find corner cases that are very difficult to find with testing alone. Hence testing and formal model checking are complementary and should not be considered as alternative means to acquire assurance in the software. Both activities results in Reports that are then merged resulting in a Development Report that together with the TestReports and Review activities results in a Release of the software.

In Figure 5, the Dependency tree generated with the planning activities is shown in a textual format. The same filtering is used as above.
Figure 4 Dependency tree for a top-level Requirement
WP-9 Project Planning
--- WP-9.1 - Development Plan
---- WP-9.1.1 - Formal Modelling
------- ITM-100 - Event TLA+ Model Implementation
------- ITM-101 - List TLA+ Model Implementation
------- WPT-2 - Formal Modelling Report
------- RES-1 - Formal Model Report Approved
--------- WPT-5 - Development Review Report
----------- RES-5 - Development Review Report Approved
-------- WP-10 - Release
---- WP-9.1.2 - Implementation
------- ITM-103 - L1.RaiseEvent_W
------- ITM-104 - L1.RaiseEvent_W
------- ITM-102 - L1.List_isEmpty
------- WPT-3 - Implementation Report
------- RES-2 - Implementation Report Approved
------- WP-6 - Development Review
------- RES-5 - Development Review Report Approved
-------- WP-10 - Release
--- WP-8 - Verification
------ WPT-9 - Verification Report
------- RES-3 - Verification Report Approved
------ WP-7 - Test Review
------- WPT-6 - Test Review Report
-------- RES-6 - Test Review Report Approved
--------- WP-10 - Release
------- ITM-5.3.1 - L1.List_HeadElement
------- ITM-9.2.2 - L1.List_insertPacket
------- ITM-9.2.1 - L1.List_insert
------- ITM-17 - L1.insertPacketInKernel
------- ITM-17.1 - L1.enterCriticalSection
------- ITM-9.2 - L1.List_insertPacket
------- ITM-9.2.1 - L1.List_insert
------- ITM-17.3 - L1.List_removePacket
------- ITM-9.3 - L1.List_remove
------- ITM-17.4 - L1.leaveCriticalSection
------- ITM-17.5 - L1.Reschedule
------- ITM-17.5.1 - L1.DescheduleKernel
------- ITM-17.5.2.2 - L1.switchContext
------- ITM-9.4.1.1 - L1.restoreStatusRegister
------- ITM-17.5.2 - L1.Switch2Kernel
------- ITM-9.4.1.1 - L1.restoreStatusRegister
------- ITM-17.5.2 - L1.Switch2Kernel
------- ITM-17.5.2.1 - L1.saveStatusRegister
------- ITM-17.5.2.2 - L1.switchContext
------- ITM-9.4.1.1 - L1.restoreStatusRegister
------- ITM-17.3 - L1.List_removePacket
------- ITM-9.3 - L1.List_remove
------- ITM-5.3.2 - EventSyncCondition
------- ITM-5.3.8.3.2 - L1.isSendPacket
------- ITM-5.3.8.2 - L1.IsHubEventSet
------- ITM-5.3.8.3 - EventUpdate
------- ITM-5.3.8.3.1 - L1.IsEventHub
------- ITM-5.3.8.3.2 - L1.isSendPacket
------- ITM-5.3.8.3.3 - L1.Event_State
--- WP-9.2 - Testing Plan
---- WP-9.2.1 - Testing
------ WPT-4 - Testing Report
------ RES-4 - Testing Report Approved
------ WP-7 - Test Review
------ WPT-6 - Test Review Report
------- RES-6 - Test Review Report Approved
-------- WP-10 - Release
------- ITM-72 - UT_Event_Update_ReceivePacket
------- ITM-73 - UT_Event_Update_SentPacket
------- ITM-74 - UT_Event_SyncCondition_ReceivePacket_unset
------- ITM-75 - UT_Event_SyncCondition_ReceivePacket_set
------- ITM-76 - UT_Event_SyncCondition_SentPacket_unset
------- ITM-77 - UT_Event_SyncCondition_SentPacket_set
------- ITM-78 - UT_Event_SyncCondition_noModify_ReceivePacket_unset
------- ITM-79 - UT_Event_SyncCondition_noModify_ReceivePacket_set
------- ITM-80 - UT_Event_SyncCondition_noModify_SentPacket_unset
------- ITM-81 - UT_Event_SyncCondition_noModify_SentPacket_set

Figure 5 Project Planning Dependency Tree
3.1.3 Data exchange with the OPENCOSS platform

While considering the data generated or present in a GoedelWorks portal, next arises the question of how this data can be mapped onto the CCL model of the OPENCOSS platform.

Considering the Hub Event subset of the complete OpenComRTOS project dataset, 273 entities were counted as relevant for the project.

The following types of artefacts are available:

- Documents:
  - Planning documents
  - Design documents
  - Test reports
  - Verification reports
  - Review reports
  - A .html or .pdf dump of the full project (meant as a snapshot).

- Requirement statements

- Specification statements:
  - RTOS services
    - As part of the generated user documentation from the source code
    - As API specification using C syntax.
  - TLA+ specifications
  - Proven invariants during formal model checking
  - Test specifications

- Traceability trees
  - Graphical (e.g. .png or on-line) or as text files
  - Issue tracking system tickets

- Items
  - Source code (.C and .h files) via links to a svn repository
  - TLA+ models via links to a svn repository.

3.1.3.1 Data exchange possibilities with the OPENCOSS platform

At this stage a clear mapping between the GoedelWorks artefacts and the CCL defined entities is not yet possible. Using the interim CLL defined in D4.3, we attempt a first mapping to the CCL terms. The four most important CCL terms are at the moment of writing this report (status: 11.05.2013):

1. Activity
2. Artefact
3. AssuranceAsset

The corresponding entities in a GoedelWorks project are:

1. Activity: Work Packages and Tasks
2. Artefact: Work Products, such as developed Items and Documents (e.g. Reports)
3. AssuranceAsset: Resources (personel with a role, equipment, etc.)
4. AssertedEvidence: dependency trees, justifying the Approval state of a project entity.

Ignoring the attributes, an import/export or data exchange is at this stage likely to be only partially possible. While individual entities can be exchanged, it is not clear how the relationships can be exchanged. In GoedelWorks these relationships express dependencies by refinement, structural links by decomposition as well as workflow partial orders. Finally, the finally approved item (i.e. the system under development) is
only “released” (Approved in GoedelWorks terminology) when all preceding entities in the dependency
trees have been approved. It is not clear how this will be expressed in the CCL.

For the data exchange itself, two possibilities seem possible:

1. Automation: e.g. by generating intermediate description files referring to the GoedelWorks files
   (dumped as .pdf files). In each group of entities of the same type, CSV is another option. This
   assumes that automatic mapping is possible. The latter is possible using the OSLC approach (see
   further).
2. Manual: a human user uses an OPENCOSS import/export panel, describing and assigning
   OPENCOSS CCL terms and attributes to the imported files.

We refer to section 5 for a potential solution using OSLC.
3.2 Evidence in the Parasoft products portfolio (PSF)

3.2.1 Brief overview of the Parasoft products

Parasoft has researched on and developed software solutions that help organizations deliver defect-free software efficiently. By integrating end-to-end testing, dev/test environment management, and software development management, we reduce the time, effort, and cost of delivering secure, reliable, and compliant software.

Parasoft provides a centralized, comprehensive solution to achieve the functional safety goal recommended by ISO 26262&ASIL, including static code analysis, automated unit testing, coverage analysis, traceability, component and regression testing. Parasoft is recognized by software development professionals as a leader in software development lifecycle automation.

Figure 6 shows a typical deployment of Parasoft Concerto as a central element of the ALM ecosystem:
The “automatic test servers” element in Figure 6 represents our broad offer of testing tools: Jtest, C++test, .Test, and SOAtest. These tools provide a means for establishing code quality measurement metrics, and therefore serve to create “evidence artefacts” for the kinds of high-integrity systems with which OPENCOSS is concerned (e.g., coding standards compliance, unit tests results and coverage metrics, functional tests of remote interfaces, code review process efficiency etc.). Moreover, by using our products, an interested party can access SCM and BTS activity, status of manual test scenarios execution, and other strategically important project health factors.

The exact role that our tools can play as building blocks of the OPENCOSS platform were described in detail in Appendix E of the D6.1 final document. Please read it at your convenience. We will skip it here to prevent repetitions.

One important thing to emphasize here is that, due to the huge integration capabilities and the broad spectrum of information available in our products’ ecosystem, we can provide the basis for the transitive evidence capabilities that would build the evolutionary evidence chain. For example, an interested party can navigate between our artefacts and be able to observe how single code commit influences a related task/BTS artefact/requirement. Automatic tests and manual test scenarios should be re-run to test the code parts affected by these changes and so on.

### 3.2.2 Evidence types suitable for the OPENCOSS platform

#### 3.2.2.1 Static code analysis results

Our tools can run static analysis against the codebase to enforce certain good coding practices, such as establishing consistent coding standards, performing check against common coding errors defined by well-established standards, and advanced symbolic simulation of code flow.

These results are sent and stored in Concerto for further analysis in wider perspective but can also be exported to XML files. Below there is a snippet of such a report, in which “(...)” means repetitive blocks cut from the example.

```xml
<ResultsSession time="04/25/13 15:58:25" toolName="Jtest"
toolVer="9.5.0">
  <TestConfig machine="opeth" name="Static Analysis" pseudoUrl="jtest.builtin://Static Analysis" user="dariuszo"/>
  <Authors>
    <Author id="dev1" name="dariuszo"/>
  </Authors>
  <VersionInfos>
    <StorageInfo ownerId="com.parasoft.xtest.checkers.api.execution"
      resultId="IExecutionViolation" version="2"/>
    <Authors id="dev1" name="dariuszo"/>
  </VersionInfos>
  <Scope>
    <ProjectInformations>
      <ScopeProjectInfo fltFiles="1" fltLns="11"
        project="/com.parasoft.jtest.springtestproject" totFiles="1" totLns="11">
        <File path="src/main/java/com/parasoft/jtest/springtestproject/InjectableBean.java"/>
      </ScopeProjectInfo>
    </ProjectInformations>
    <CodingStandards ownerId="com.parasoft.xtest.checkers.api.standards"
time="0:00:00">
      <Goal mode="1" name="Static"/>
      <Projects>
        <Project bdCheckedFiles="0" bdTotalFiles="0" checkedFiles="1" checkedLns="11"
          name="/com.parasoft.jtest.springtestproject" qfixErrs="0" suppErrs="0" totErrs="1"
          totFiles="1" totLns="11"/>
      </Projects>
    </CodingStandards>
    <StdViols>
      <StdViol save="1" ln="3" cat="NEED" hash="-1156915665" urgent="true" tool="jtest" locType="sr">
        <msg>Bean class does not implement 'java.io.Serializable': InjectableBean</msg>
        <lang>java</lang> rule="BEAN.SERIALIZABLE" config="2" auth="dariuszo"
        locOffs="60" locLen="14"
        locFile="(...)InjectableBean.java"/>
    </StdViols>
  </Scope>
</ResultsSession>
```
3.2.2.2 Unit tests results

Our tools can manage and perform unit tests. These can be either automatically generated by our tools to establish a regression tests harness or to test code quality (e.g., if public APIs are written in a proper way) to provide a bulletproof interface against typical abuse (insufficient input, sanity checks, etc.). Customer-provided hand-written unit tests can be run by our tools too, providing unified access and documentation of the whole testing process.

These results are sent and stored in Concerto for further analysis in wider perspective but can also be exported to XML files. Below there is a snippet of such a report, in which (...) means repetitive blocks cut from the example.

```
<xml version="1.0" encoding="utf-8"/>
<ResultsSession time="04/25/13 16:06:37" toolName="jtest" toolVer="9.5.0">
  <TextConfig machine="opeth" name="Generate and Run Unit Tests" 
    pseudoUrl="/generateAndRunUnitTests" user="dariuszo" />
  <Authors>
    <Author id="dev1" name="dariuszo" />
  </Authors>
  <VersionInfo>
    <StorageInfo ownerId="com.parasoft.test.checkers.api.standards" resultId="IExecutionViolation" ver="2">
      (...)
    </StorageInfo>
  </VersionInfo>
  <Scope>
    <ScopeProjectInfo path="/com.parasoft.jtest.springtestproject/InjectableBean.java" />
  </Scope>
  <Generation time="01:02:11">
    <Projects>
      <GenerationProjectInfo filesChecked="1" 
        filesSkippedAsTests="0" filesSkippedWithOldTests="0" filesSkippedWithoutTests="0" project="/com.parasoft.jtest.springtestproject/InjectableBean" 
        testCase="2" />
    </Projects>
    <GenerationInfo auth="dariuszo" 
      fileType="/com.parasoft.jtest.springtestproject/InjectableBean" 
      time="01:02:11" unitTestType="/com.parasoft.jtest.springtestproject/InjectableBean" />
  </Generation>
</ResultsSession>
```
3.2.2.3 Code review results

A Code review client is integrated into our language tools, which can be run along, for instance, Jtest in the Eclipse environment. The state of the code review processes, statistics and reports are available in Concerto. Communication between clients and Concerto is performed through a SOAP interface.

3.2.2.4 Manual tests results

Concerto can store manual test scenarios, as well as the results of running these scenarios by the QA team. These scenarios can be assigned to requirements, tasks, or defect reports, and provide a way to verify the effort spent on these artefacts. This can be useful when a need to perform cross-system tests in whole modules related to latest code commit arises.
3.2.2.5 Tasks/defects/requirements state
Concerto scans connected BTSes but has its native issues/tasks/requirements tracking system. All this data is available in a uniform way along with statistics and quality metrics. These could be accessed using a SOAP/REST API, should such a need arise.

3.3 Evidence in medini analyze (IKV)

3.3.1 General overview on the tool
medini™ analyze is a toolset supporting the safety analysis and design for software controlled safety related functions in cars. It is specifically tailored to ISO 26262 and integrates system architecture design (SyML) and software functional design (MATLAB®/Simulink®/Stateflow®) with risk and hazard analysis methods - Hazard List, Risk Graph, Fault Tree Analysis (FTA) and Failure Mode and Effects Analysis (FMEA).

Moreover medini™ analyze provides a central management of all the safety goals and corresponding safety requirements. Being built on top of Eclipse technologies like EMF, medini™ analyze traces and tracks all safety relevant information and decisions throughout the whole development process.

The following table shows the major methods that are integrated by medini analyze.

<table>
<thead>
<tr>
<th>Hazard analysis, Risk assessment and ASIL determination</th>
</tr>
</thead>
<tbody>
<tr>
<td>▪ table-based framework to manage all hazardous events</td>
</tr>
<tr>
<td>▪ determination of Automotive Safety Integrity Levels (ASIL) according ISO DIS 26262</td>
</tr>
<tr>
<td>▪ profiling to other standards such as IEC 62304 and DO 178B possible</td>
</tr>
<tr>
<td>▪ re-use known driving situations and hazards</td>
</tr>
<tr>
<td>▪ fine grained traceability to detailed safety analysis and safety goals</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Safety goal analysis and management</th>
</tr>
</thead>
<tbody>
<tr>
<td>▪ integrated graphical safety goal and requirements model editor</td>
</tr>
<tr>
<td>▪ use safety goal models as a means to capture and manage high-level safety requirements and ASIL requirements</td>
</tr>
<tr>
<td>▪ derive functional and technical requirements from safety goals</td>
</tr>
<tr>
<td>▪ evaluate the safety goal model to find out contradictions and to detect alternative solutions e.g. through ASIL decomposition</td>
</tr>
<tr>
<td>▪ fine grained traceability to system model, software model and safety analysis</td>
</tr>
</tbody>
</table>
System architecture modeling
- integrated graphical architecture model editor
- specification of the high-level system architecture using SysML
- specification of failure modes and failure rates for elements of the system architecture
- profiling to AUTOSAR methodology and notation guide
- traceability to elements of the software model and safety analysis such as FTA and FMEA

Failure Mode and Effects Analysis (FMEA)
- provision and customization of standard templates for FMEA
- tables are by default already populated with the components/functions from the system models
- failure modes of elements of system model are automatically added
- automatic computation of Risk Priority Numbers (RPN) to prioritize which items require additional quality planning or action

Fault Tree Analysis (FTA)
- integrated FTA editor enables the creation of fault trees and their quantitative evaluation
- semi-automatic population with failure events from the system architecture model (including failure rates)
- determination and evaluation of minimal cut-sets to find out their probability and importance
- seamless navigation from cut-sets to elements of the system design (based on traces)
- re-calculation of probabilities and importance after design changes
3.3.2 Supported evidence types

Due to its nature the methods produce a rich set of evidence types. These evidence types are described as follows:

3.3.2.1 HARA

The tool can be used to perform hazard analysis and risk assessment. As a result an arbitrary number of hazard lists including considered situations, risk classification and links to derived safety goals can be created and maintained in the tool. These hazard lists can be imported and exported using de facto standard table exchange formats such as CSV or Excel.

3.3.2.2 FMEA/FTA

FMEA and FTA can be executed based on an architecture model or standalone. The results are full fledged FMEA/FMEADE worksheets and fault trees including failure modes, links to architectural elements and their properties. For the FTA also the graphical notation as well as cut-set calculations is available.

3.3.2.3 Goals and Requirements

The tool supports rich requirement engineering and the breakdown of goals to contributing requirements and composition as defined by ISO 26262. Requirements can be interchanged with other tools as IBM Rational DOORS or tools supporting RIF. The engineering is supported by a graphical notation.

3.3.2.4 Architecture

Architecture models are the basis for most other activities performed in the tool although not absolutely necessary. Models can be created in the tool using a subset of the SysML language and notation or imported from other external SysML tools as Rational Rhapsody or Enterprise Architect if they are already available as models. The imported models are treated as snapshots and thus read-only copies. Certain safety data that have been added in the tool (e.g. results from FMEA, FTA or reliability engineering) are stored in the architecture model though. The architecture can be updated at any time and merged with the local copy.

3.3.2.5 Traces

While performing the different activities a rich set of trace relations between all types of artefacts is created either automatically as effect of the activity or explicitly by the user. For example when allocating functions or requirements to the system architecture. These relations are available at each artefact and can be navigated to find relation paths between two arbitrary artefacts (e.g. shortest path) or to calculate change impacts.

3.3.2.6 Supporting artefacts

Besides the standard evidence types coming from certain safety activities there is a set of generic artifacts that are typically used to support the evidence or argumentation like additional documents or checklists.

3.3.3 Potential data exchange within OPENCOSS

The tool supports a set of point-to-point integrations with proprietary tools such as IBM rational DOORS or Enterprise Architect to achieve a smooth integration with the existing tool landscape especially in the automotive domain. Most of these tools were integrated via their proprietary Windows APIs (COM, .NET) or via standardised XML files as RIF.

The medini analyze tool is using eclipse and EMF technology to store all kind of artefacts. The stored data can be directly accessed or exchanged with other tools that are using the same meta-models via XMI. Alternatively additional plugins can be developed atop the meta-models and the EMF Java API to access or manipulate the data at runtime.
Experiences with these proprietary point-to-point integrations have shown that it’s not cost effective. Even standardized exchange formats as RIF do not necessarily reduce the integration effort if they lack support by tool vendors or if semantic details are underspecified. The complete import of data allows self-contained “safety cases” but on the other hand leads to update and merge problematic later and to adaptation to local data structures. New (old) approaches like OSLC propose trying to keep data were they are but instead to reference them only. There are still many open questions, that have to be evaluated in detail (e.g. baselines etc.) but the approach itself is interesting for upcoming work on the tool.

4 Initial ideas for an integration API as seen by ParaSoft

In this section, we list some general suggestions as to how OPENCOSS can be integrated with external tools. This reflects the view of ParaSoft as a tool vendor. The intent is to encourage discussion on the subject and to establish a common ground within the project as a whole.

4.1 Establishing a common registry of evidence types provided by external tools

In the evidence taxonomy (see Figure 7), there is a list of types of evidence. Let’s use Parasoft’s testing tools as example. According to paragraph 3.3.2 of this document these tools can produce evidence of types such as “Automated static analysis results” and “Unit Testing Results”, amongst others. Every evidence artefact gathered from every tool supporting OPENCOSS export should provide an evidence classification type, as well as some basic properties specific for this evidence type.
Figure 7 Evidence taxonomy
4.2 Establishing a common format for evidence import/export from external tools

This format can be based on XML. Such a XML document can consist of an envelope, which would contain some well-established set of attributes such as evidence type (see Section 4.1), state information adequate to a given evidence type (e.g. “passed” or “no errors” for “Automated static analysis results” type), some standard attributes like tool name and version, and document origin information. Additionally, such documentation should contain a payload containing original, native tool report/log, which would be treated as a proof and could be inspected on demand at a given point in time.

What is important here is that we should establish a minimal set of envelope attributes, so that it would be sufficient to process only these few attributes by a reasoner (see the Semantic-Technologies-related ideas in D6.3) or similar means, and make statements on the condition of an evidence chain.

4.3 Data integrity assurance

The common format mentioned in section 4.2 should take care of data integrity control. Evidence, once created and exported by an external tool, should be secured against data tampering. This could be accomplished by signing the envelope and payload and by storing it along the evidence.

4.4 Unique identifiers for every imported evidence

It should be possible to address all evidence by a unique identifier. Evidence import/export facilities should take care of it. Additionally, a way to translate these GUIDs to native tools UIDs can be required, so that we could address resources/evidences available on satellite systems.

4.5 Pull + push import/export

Some tools with which we may want to integrate have workflows that will naturally lead to explicit data export performed as a result of action (test/inspection) performed. Yet there are tools for which such a push approach to data export will be not sufficient. This can be a BTS providing info on a specific defect, or Concerto providing information about the current state of a software project (state of code review, state of manual tests, state of requirements addressed, test failures metrics, etc.).

To address these facilities, we could establish a service that would “understand” how to deal with such remote evidence providers (e.g., knowing how to query Concerto on a specific aspect of project condition.)

4.6 Relation between imported/remote evidence traced by unique identifiers

Unique identifiers mentioned in section 4.4 can be used to express relations between evidences, even among different remote systems from different providers. Such a relations organization could allow express dependency between evidences that could be navigated automatically (or at least organized in a well-defined way).
5 Overview of OSLC as data exchange protocol

5.1 Short introduction on IFEST

Some guidelines for exchanging data between external tools and the OPENCOSS platform as tool can be obtained from the ifEST project [iFEST]. The ifEST project (industrial Framework for Embedded Systems Tools) aims at specifying and developing an tool integration framework for HW/SW co-design of heterogeneous and multi-core embedded systems. While the target domain is different from the certification domain of OPENCOSS, the technical goals are very similar.

IFEST largely aimed for by specifying and developing so-called Adaptor tools using the OSLC specifications [OSLC1]

5.2 Short introduction on OSLC

Open Service for Lifecycle Collaboration (OSLC) is an open community, originally proposed in 2008 [OSLC0], to define a set of specifications that enable integration of software development and more broadly Application Lifecycle Management (ALM) and Product lifecycle Management (PLM) products and services. The intention is to make life easier for software and product developers and tools vendors, by making it easier for tools to work together [OSCL2]. While the target domains are different, there are overlapping domains (e.g. Requirements Management) and the challenge is similar to the one in OPENCOSS, whereby data from external tools and environments must be integrated, imported or exported without losing essential information.

5.3 Guidelines from the IFEST project and the OSLC community

The ifEST project has produced guidelines that show the complexity of the task is fairly large. From the project report we deduce the following guidelines by transposing it into account the OPENCOSS context:

Before specifying a Tool Adaptor for a specific tool there is some background research that must be completed. The developer must have some expert knowledge of the tool to be implemented, which means he must fully understand the semantics and be aware of semantic differences as well as the methodology behind it. He must have some idea of which CCL specification is to be implemented. He must have a list of integration scenarios that are relevant to the tool. Armed with this knowledge he will then ask himself a number of questions:

- What data do I need to expose to other services?
- What data do I need from other services?
- Is the Tool Adaptor for the tool I am about to implement similar to another, that is already OSLC compliant?
- Can I export data from the tool via the Tool Adaptor using a standardized interchange format?
- Does the tool server comprise a repository?
- Can I use an existing OSLC specification?
- Can I use an existing CCL specification?
- Should I create a new OSLC specification?
- Should I create a new CCL specification?
We notice that the iFEST project approach favours the use of the OSLC specifications as an intermediate representation. However, using OSLC will require a careful study of the domains now available for each external tool as well as between CCL and OSCL. We notice also that OSLC specification is very much syntax driven, focusing on data exchange, and less by semantic properties. The latter are quite important for assurance and evidence management. Therefore further study is needed.

There are (broadly) three different approaches to implementing an OSLC based data exchange [OSLC3]:

1. The **Native Support** approach is to add OSLC support directly into the tools, modifying whatever code is necessary to implement the OSLC specification.
2. The **Plugin** approach is add OSLC support to the tool by developing code that plugs-in to the tool and uses its add-on API.
3. The **Adapter** approach is to create new web application that acts as an OSLC Adapter, runs alongside of the software tool, provides OSLC support and “under the hood” makes calls to the tool web APIs to create, retrieve, update and delete resources.

Although any of these approaches are valid approaches for an OSLC implementation, here are some of the pros and cons of each:

<table>
<thead>
<tr>
<th>Approach</th>
<th>Pros</th>
<th>Cons</th>
</tr>
</thead>
</table>
| Native approach | • Complete control over the quality implementation  
• Good approach for tool vendors shipping products with OSLC support | • Control needed over the application code  
• Need to learn the product's language and platform  
• Not a good approach for customers who want to add OSLC support to a vendor's products |
| Plugin approach   | • Uses established and supported mechanism to extend product and add OSLC support | • Limitations on plugins may limit quality of OSLC implementation  
• Need to learn product's language, platform, and plugin architecture |
| Adapter approach   | • Can be implemented without modifying the product  
• Can use the preferred platform and language | • Limitations of product's API may limit quality of OSLC implementation  
• May introduce redundant URL for product resources. |
In short, the Native approach is the right approach for tool vendors who want to add OSLC support to the products that they understand well. The Plugin and Adapter approaches are best when adding OSLC support to a tool that has been bought from a tool vendor or obtained from an open source project.

5.4 Consequences of selecting OSLC

OSLC was originally developed in the context of (semantic) web technologies and aimed at allowing the specification of “links” between data on the web. This allows retrieving more information from the data as relationships can be discovered between specific data instances. The links really express relationships between data entities as found in e.g. relational database schemes but without the presence of a structured relational database. What OSLC defines is vocabularies or ontologies for data entities and their attributes that are standardised. OSLC statements then make use of the so-called URI (Universal Resource Identifier) and RDF (Resource Description Framework) to package such data and the relationships between them in a neutral format so that OSLC enabled tools can exchange data (as far as the data represents semantically the same thing).

While the OSLC community defines core classes of terms, for specific sub-domains workgroups are formed that define the OSLC vocabulary for that specific domain, whereby each URI must be approved.

Therefore, if OPENCOSS wants to demonstrate the use of OSLC, a new OSLC workgroup needs to be set up that in particular looks at the safety and certification domain. The ontology defined in this OSLC workgroup will however be universal and open for anyone to use, so it will most likely be a superset of the CCL definition of OPENCOSS. However this subset should map onto the CCL definition to make it usable in the OPENCOSS project. At the same time, the whole OSLC framework is rather complex to set up and the supporting documentation is not always very precise. Hence, using OSLC from the very beginning is rather a heavy task that could clutter the conceptual thinking. Therefore a 3 steps approached is to be considered:

1. In a first step CCL is further developed and selected external tools are more deeply analysed in terms of the concepts and vocabulary used. No specific data format is needed.
2. In a second step, when the concepts have stabilised, an internal workshop is established that defines an OSLC compliant vocabulary.
3. In third step when the OSLC definitions are stable and proven in real use cases, the OPENCOSS contributions, eventually covering multiple certification sub-domains are proposed as addenda to the OSLC standard.
6 Conclusions

D6.4 has provided an initial analysis of how data can be exchanged between external assurance tools (and their environments) and the OPENC OSS tool platform. The deliverable has shown that at least a partial data exchange should be possible. At this moment this needs further effort, and a better clarification depends on the finalisation of the CCL. Given that the CCL develops incrementally as more experience is gained, the data exchange possibilities will evolve, and this document will be updated as well.

Although for prototyping purposes the effort can be seen as quite large, it makes sense to further study the OSLC approach. However, this means that the OPENC OSS platform and CCL should take this into account from the very beginning as a common terminology (with common semantics) must be obtained.
7 References

[ARRL] From Safety Integrity Level to Assured Reliability and Resilience Level for composable safety critical systems. Altreonic 2013, draft white paper.


[OSLC1] http://open-services.net/


Appendix A. Selected tools used by OPENCOSS partners

This information is based on the baseline survey and the interviews conducted in the scope of D3.1, with additional input by the project partners.

8.1 Tools used by project partners

Used by ALS
1. Atego Artisan Studio
2. Word
3. Excel
4. DOORS
5. Reqtify
6. GesDoc
7. ClearCase

Used by CRF
8. ADONIS

Used by TAV (Valence)
9. Mercurial
10. GNAT Pro
11. Excel
12. ClearCase
13. DOORS
14. Word
15. FIT
16. Reqtify
17. Rhapsody

8.2 Other tools developed/commercialized by other partners

ADACORE
18. GNATcheck
19. GNATcoverage
20. CodePeer
21. GNATPeer
22. GNATTest
23. The Qualifying Machine (probably to be integrated into the OPENCOSS tool platform)

ALTREONIC
24. GoedelWorks

ATF & ATU
25. Atego Artisan Studio
26. Atego Process Director
27. Atego Workbench

IKV
28. Medini Analyze

PSF
29. Parasoft Concerto
30. Parasoft C++test
31. Parasoft Jtest
32. Parasoft .Test

SiM (not completely sure if integration or reuse is possible, because of IP issues)
33. Modus
34. CRESCO
35. PROCE
36. EvidenceAgreement
37. SafeSlice

TUE
38. ProM
39. TraceVis
40. FRASR
41. mCRL2
42.

8.3 Other possible tools

38. ClearQuest
39. Matlab/Simulink
40. Safety/assurance case editors (ASCE, Certware, etc.)
41. Scade
42. VectorCasr
43. GCover
44. IBM Rational tools
45. CodeTEST
46. Design Verifier
47. IPL tools
48. Advitium
49. LDRA tools
50. Polarsys