Engineering tools for creation, integration and maintenance of DDIs V2

White Paper

www.deis-project.eu

This project has received funding from the European Union’s Horizon 2020 research and innovation programme under grant agreement No 732242 (DEIS).
Table of Contents

1 Introduction ........................................................................................................................................... 4

2 Background ............................................................................................................................................... 5
  2.1 The Epsilon Language Framework ................................................................................................. 6
  2.2 ANT Build System ............................................................................................................................. 8

3 DDI Tool Framework ............................................................................................................................. 8
  3.1.1 Script Storage/Management ......................................................................................................... 8
  3.1.2 Script-based Operations ............................................................................................................... 9
  3.1.3 Script Execution Orchestration ..................................................................................................... 9
  3.1.4 External Service Invocation ......................................................................................................... 15

4 Semi-Automatic DDI Case Study ........................................................................................................... 17
  4.1 ETCS .................................................................................................................................................. 17
    4.1.1 Semi-Automatic DDI Integration .................................................................................................. 18
    4.1.2 DDI Dependability Analysis via External Service Requisition .................................................. 20
    4.1.3 Verification of Process & Product-Related Dependability Requirements .................................... 22
    4.1.4 Automatic Generation of Dependability Evidence based on DDI Analysis Results ............... 26

5 Further Discussion and Conclusion ..................................................................................................... 29

6 References ............................................................................................................................................. 30
List of Figures

FIGURE 1 - EXAMPLE MODEL BEFORE EOL EXECUTION ................................................................. 7
FIGURE 2 - EXAMPLE EOL SCRIPT .......................................................................................... 7
FIGURE 3 - EXAMPLE MODEL AFTER EOL EXECUTION .......................................................... 7
FIGURE 4 - SAMPLE OF EVL CONSTRAINTS ........................................................................... 7
FIGURE 5 - GENERIC CREATION AND EXECUTION OF EPSILON DDI SCRIPTS WITHIN THE DDI TOOL ADAPTER ................................................................. 10
FIGURE 6 - DDI TOOL ADAPTER THRIFT SERVICE INTERFACE ........................................... 11
FIGURE 7 - DDI TOOL ADAPTER THRIFT SERVICE INPUT SETUP FOR DDI SCRIPT EXECUTION .......................................................................................... 12
FIGURE 8 - ABSTRACT EXAMPLE OF ANT W/ EPSILON USAGE ........................................... 14
FIGURE 9 - EXTERNAL SERVICE INVOCATION CONCEPT ..................................................... 15
FIGURE 10 - EXTERNAL SERVICE INVOCATION VIA EPSILON SCRIPT ................................ 16
FIGURE 11 - EXTERNAL SERVICE PROVISION FOR USAGE IN EPSILON SCRIPT .................. 16
FIGURE 12 - OVERVIEW OF ETCS WITH SEMI-AUTOMATIC INTEGRATION ............................ 17
FIGURE 13 - SACM INTEGRATION EOL SCRIPT ..................................................................... 19
FIGURE 14 - DESIGNPACKAGE INTEGRATION EOL SCRIPT .................................................... 19
FIGURE 15 - REFERENCING ELEMENTS IN DIFFERENT DDIs ............................................... 21
FIGURE 16 - EPSILON EOL SCRIPT FOR ANALYZING A TOP EVENT WITH AN EXTERNAL TOOL SERVICE .............................................................. 22
FIGURE 17 - INVOLVED DDI ELEMENTS FOR CHECKING FAILURE RATE SATISFACTION REQUIREMENT ................................................................... 23
FIGURE 18 - EVL SCRIPT FOR VALIDATING QUANTITATIVE ANALYSIS REQUIREMENTS .......... 24
FIGURE 19 - ETCS PROCESS REQUIREMENTS AND MODEL ELEMENTS CONTAINING EVIDENCE ...................................................................... 26
FIGURE 20 - EVIDENCE GENERATION SCRIPT ..................................................................... 27
FIGURE 21 - EOL SCRIPT FOR ALTERNATIVE SUBSYSTEM REPLACEMENT ............................ 28

Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Long Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>CFT</td>
<td>Component Fault Tree</td>
</tr>
<tr>
<td>DDI</td>
<td>Digital Dependability Identity</td>
</tr>
<tr>
<td>DPMS</td>
<td>Dependable Physiological Monitoring System</td>
</tr>
<tr>
<td>EMF</td>
<td>Eclipse Modeling Framework</td>
</tr>
<tr>
<td>ETCS</td>
<td>European Train Control System</td>
</tr>
<tr>
<td>GSN</td>
<td>Goal Structuring Notation</td>
</tr>
<tr>
<td>ODE</td>
<td>Open Dependability Exchange Metamodel</td>
</tr>
<tr>
<td>SACM</td>
<td>Structured Assurance Case Metamodel</td>
</tr>
</tbody>
</table>
### Authors

<table>
<thead>
<tr>
<th>Name</th>
<th>Organization</th>
<th>E-mail</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yiannis Papadopoulos</td>
<td>University of Hull</td>
<td><a href="mailto:Y.I.Papadopoulos@hull.ac.uk">Y.I.Papadopoulos@hull.ac.uk</a></td>
</tr>
<tr>
<td>Ioannis Sorokos</td>
<td>University of Hull</td>
<td><a href="mailto:i.sorokos@hull.ac.uk">i.sorokos@hull.ac.uk</a></td>
</tr>
<tr>
<td>Jan Reich</td>
<td>Fraunhofer IESE</td>
<td><a href="mailto:Jan.reich@iese.fraunhofer.de">Jan.reich@iese.fraunhofer.de</a></td>
</tr>
<tr>
<td>Ran Wei</td>
<td>University of York</td>
<td><a href="mailto:ran.wei@york.ac.uk">ran.wei@york.ac.uk</a></td>
</tr>
</tbody>
</table>
1 Introduction

This document presents the second version of the engineering tools developed for supporting semi-automated synthesis, integration and evaluation of Digital Dependability Identities (DDIs). With this set of tools, users can streamline the process of producing, exchanging and applying analyses on DDIs without manual intervention at each step.

Deliverable 4.4 builds upon the progress made in deliverables:

- D3.1, where the common exchange format of the DDIs, the Open Dependability Exchange (ODE), was first described
- D4.1, where the ODE v1 metamodel technical specification was provided and discussed
- D4.2, where manual integration of DDIs with tool support was demonstrated
- D3.2, where the algorithms for semi-automated application of DDIs towards the relevant DEIS use cases was presented, alongside improvements and refinement of the ODE
- D4.3, where the technical specification of the updated ODE v2 from D4.1 was provided and discussed

The present deliverable is structured as follows. In Section 2, the relevant background for the technologies introduced to support semi-autonomous capabilities for DDIs is discussed, alongside explanatory usage examples. In Section 3, the prototypical software infrastructure to support the above capabilities is presented. In Section 4, the ETCS use case presented in D4.2 is extended to demonstrate semi-automated integration and evaluation of DDIs exchanged between an OEM and a Tier-N supplier, using the technical improvements described previously. The document concludes in Section 5 with a short discussion of the current progress and next steps.

The tools developed by DEIS, discussed in the upcoming sections 3 and 4, can be acquired and deployed from the DEIS repository, at https://github.com/DEIS-Project-EU/DDI-Scripting-Tools. Interested parties are invited to follow the online documentation to deploy the semi-automatic DDI tools to their development environment of choice.
2 Background
Per the current status of the ODE v2, described in D3.2 and provided in D4.3, a given DDI can consist of a large variety and number of supporting dependability models. While manual construction, modification and exchange of such DDIs has been demonstrated in D4.2, further tool support is necessary to address the requirements of the DEIS use cases.

Specifically, as planned in Task 4.2 of the DEIS project plan, D4.3 provides tools for:

- Semi-automated synthesis based on information and DDI fragments
- Semi-automated synthesis based on requirements from CENELEC EN (and ISO26262)
- Semi-automated integration, with the possibility for manual corrections to the subject DDIs as well
- Automated generation of assurance case argumentation, as well as impact analysis changes

Towards this end, the DEIS consortium partners identified the need to improve current tool support with respect to model management capabilities. Currently, the DDI captures a static view of a CPS, whether that represents an assurance case and/or more specific dependability assessment/assurance model(s). To gradually introduce automation to the process, programmable elements for model manipulation are needed.

Programmable elements enable the user to specify operations that can be applied en masse to a model’s elements. Furthermore, programmable elements assist in maintaining a level of abstraction that alleviates the need to address the specifics of a given model. Thus, operations can be defined generically and applied to a large range of models with little to no change necessary.

There are many existing technological solutions for introducing such programmable elements, including developing a novel one. While it would be certainly possible to extend the DEIS partner tools with such features built from the ground up, this option was not ideal for the following reasons. First, significant development work that is not directly essential to the DEIS project requirements is needed to develop the software infrastructure. Indeed, more mature software solutions already exist and have been extensively applied in similar contexts. Finally, developing a novel solution would require integration support with each of the partner tools, presenting a secondary development burden. While this burden could be partially alleviated by leveraging the common tool adapter, nonetheless, extensions for interacting with the programmable elements would have to be implemented in any partner tools that intended to support the feature.

Instead, the DEIS consortium decided to adopt model scripting languages as a more appropriate solution. Such languages offer uniformity across metamodels and backend programming languages, as they typically are defined and operate independently from the latter. An even greater benefit is that scripting languages offer tool users direct control over the model manipulation via programmable elements. Thus, the modeling tool imports dependability-related information into a model but is not responsible for performing analyses or validation on it, delegating such operations to specialist tools. In effect, the user becomes less
dependent on the choice of tool and supporting developer, which fits the role of the DDI as a technology that streamlines development barriers for CPS.

Given earlier choices made with regards to the definition of the DDI ODE metamodel and considering the cross-language mechanism embodied in the common tool adapter (see D4.2), the partners identified a specific set of technologies appropriate to serve as a solution for DEIS.

2.1 The Epsilon Language Framework

Epsilon\(^1\) (Extensible Platform for Specification of Integrated Languages for Model Management) is a collection of task specific model management languages that offer a variety of model manipulation features, such as model-to-model transformation, model-to-text generation, model validation, etc. All Epsilon languages extend the core language – the Epsilon Object Language (EOL). EOL is a general-purpose model management language, which is highly extensible. This means that new task-specific languages can be tailored by extending EOL.

In addition to the extensive support for model management languages, Epsilon also provides extensive support for different modeling technologies. Whereas the Eclipse Modeling Framework (EMF) plays a pivotal role in the modeling domain in the Eclipse eco system, Epsilon provides support to manage models not only defined using EMF but any other formats. In its current status, Epsilon supports EMF, UML, Simulink, JSON, PTCIM and many others. If needed, additional model drivers can be created to support arbitrary modeling technologies.

The Epsilon language (family) was chosen for DEIS for both of its extensive support for modeling technologies and the extensive support for model management languages. Epsilon is a mature solution that has been applied successfully in comparable problems [1-6].

The Epsilon Object Language (EOL) borrows most of its language concepts from the Object Constraint Language (OCL) [8], but provides additional features such as imperative statements, transactional model managements, and user feedback.

EOL is comparatively more flexible for “automating tasks that do not fall into the patterns targeted by task-specific languages” of the Epsilon family [7].

EOL can be used to both programmatically modify existing models, as well as generate new ones. For the purposes of the current deliverable, synthesis and integration are implemented in EOL, as it immediately addressed the needs of the use case. Further investigation into the other languages of the Epsilon family and their potential application for DEIS is ongoing.

In Figure 1, a small abstract example of applying an EOL script (in Figure 2) to generate and update a simple model can be seen. The model initially has a single system within it; through the script, a port is added to the system. The resulting model is seen in Figure 3

\(^1\) https://www.eclipse.org/epsilon/
For DDI validation, the Epsilon Validation Language (EVL) has already been applied in D4.2, to ensure that constructed DDIs are neither incomplete (e.g. references to missing elements) or incoherent (e.g. an input port producing output failures). A sample of the constraints specified can be seen in Figure 4.

The ‘PortWithInputFailure’ constraint specifies that any input port (having a ‘direction’ attribute set to ‘IN’) must only have in its ‘interfaceFailures’ property only ‘InputFailure’-type instances. The ‘PortWithOutputFailure’ constraint specifies a similar constraint for output ports and output failures.
‘PortIsConnectedToOtherPort’ checks whether a port is unlinked, which indicates there might be an oversight with the architecture’s design or implementation.

2.2 ANT Build System

While useful on their own, model manipulation/evaluation via scripts can be further enhanced when paired with a task automation framework. Such a framework can organize the cooperation between multiple scripts, order and establish conditions on the invocation of specific scripts and dynamically adapt the process according to the state of the subject model(s).

Apache Ant\(^2\) is a task automation system typically employed as a programming language ‘build’ system, with a focus towards Java. Build systems compile source code into linkable code units, then link code units together as well as with common library dependencies they may have, to produce the target executable program. Additionally, build systems typically configure the build process according to the host and target machine; the former performs the build process, whereas the latter is the platform where the executable is intended to be deployed on.

Beyond building, Apache Ant can be employed to automate other tasks related to software, such as program testing. Furthermore, it can also be used in conjunction with Epsilon to automate the loading and storing of models and execution of Epsilon scripts on them.

Ant operates on a user-defined ‘buildfile’, written in XML. A buildfile consists of a single ‘project’ element and at least one default ‘target’ element. A target consists of one or more ‘task’ elements. When Ant executes against the buildfile, if no targets were specified, it will begin by executing the tasks defined in the default target. The target under execution can depend on the execution of other targets; this leads to the execution of the target’s dependencies (i.e. further targets), the failure of which may cause the process to adjust or abort.

3 DDI Tool Framework

In this section, the technical progress made across the DDI engineering toolchain is described. Emphasis is given towards the features supporting semi-automated synthesis and evaluation of DDIs.

3.1.1 Script Storage/Management

Scripts are currently stored independently from the DDIs they are executed upon. Users develop the scripts using the toolchain of their choice. For Epsilon scripts, the typical toolchain is the Eclipse IDE, using the Epsilon plugin, available as a standalone distribution online\(^3\). However, as Epsilon scripts are human-readable text, users can even develop with just basic text editors.

---

\(^2\) https://ant.apache.org/index.html

\(^3\) https://www.eclipse.org/epsilon/download/
3.1.2 Script-based Operations
The DEIS partners identified the following initial semi-automated services necessary to further the capabilities of the DDI with regards to D4.4:

- Semi-Automatic Import and Export of DDIs. Integration of DDIs requires seamless acquisition, merging and sharing capabilities.
- DDI Constraint Validation. This service has already been demonstrated as a supported feature in D4.2; in D4.4, the feature can now be invoked without user interaction.
- Safety Case Fragment Generation. Supporting dependability assurance is an important aspect of the DDI, thus the provision of a safety case for the subject CPS is also supported with semi-automation. For D4.4, this service is demonstrated through an OEM-Tier-N-Supplier scenario, via a top-down approach.
- Tool-Specific Service Requisition. The toolchain available to a partner in the OEM-Tier-N development lifecycle (but not necessarily to all), may provide unique dependability assessment/assurance services. This service enables semi-automatic requisition of such services and incorporation of their results within the DDI.

3.1.3 Script Execution Orchestration
Although a single script execution might be sufficient to satisfy a given use case, in general, multiple scripts potentially defined in different languages may be required to perform all the necessary operations. Furthermore, the order and conditionality of the above script execution may also be factors that need to be addressed to maintain an execution flow without requiring frequent user interaction.

In the previous iteration step of the DEIS tool adapter (documented in deliverable D4.2), the focus rested on the import and export of models from modeling tools into the DDI format and vice versa, to ease the exchange of DDIs. Import and export were previously the only services provided by the DDI tool adapter to tool providers (such as the consortium modeling tools safeTbox, HiP-HOPs or ComposR) via the Thrift Service Interface. The next stage of features provides support to dependability engineering by improving task efficiency and reducing errors. Epsilon scripts were identified as a suitable means to express these automation “recipes” modularly.

A hierarchy of scripts can be established, with respect to their level of application. Simpler, lower-level scripts, such as comparing properties in different parts of the model, are reused within higher-level scripts expressing concrete activities in the dependability engineering lifecycle. An example of a higher-level script would be checking normative requirements (modeled as claims in the assurance case) for the existence of certain documentation in the form of models that are needed to satisfy the requirements. Similar to manual activities executed by a dependability engineer in sequence to comply with a higher-level process, the basic DDI scripts need to be executed on a DDI model in sequence to achieve the desired final work product, again in the form of a DDI.

The DEIS partners have identified two alternative approaches with regards to Epsilon script execution; the first one consists of an extension to the common tool adapter developed by the DEIS partners, using the
Thrift middleware library, described in D4.2. The second is based on the Apache Ant framework presented earlier, in Section 2.2.

### 3.1.3.1 Epsilon via Thrift

The high-level architecture of the DDI tool adapter for enabling the sequential Epsilon DDI script execution on given DDI models is illustrated in Figure 5. Compared to the initial version of the DDI tool adapter described in deliverable D4.2, there are three additional components:

1. The Epsilon script files, which exist as text-based files on the local machine and can be created by the dependability engineer or reused from a script library. These can be seen collected under ‘DDI Scenario X’ at the lower part of Figure 5.
2. The right part in the ‘DDI Tool Adapter’ represents the component being able to read DDI-compliant EMF models (either from DDI files or from in-memory representations of the DDI in EMF format). This component can then provide models as input to the Epsilon engine for each Epsilon script type. The Epsilon engine manages the locally provided script files and their execution on the provided DDI models.
3. Additional Thrift services callable from an external modeling tool for configuring, which local Epsilon scripts should be executed in which sequence by the tool adapter and on which DDI models the scripts should operate.

![Diagram](image)

**Figure 5 - Generic Creation and Execution Of Epsilon DDI Scripts within the DDI Tool Adapter**

To that end, we extended the DDI tool adapter’s Thrift service interface with two new services `ExecuteEpsilonScriptsOnDDIFile()` and `ExecuteEpsilonScripts()` (see Figure 6, lines 16 and 22) that take an ordered list of Epsilon scripts as part of the `TDDIServiceConfig` and execute them on the provided DDI model (either defined as file path on a local machine in the `TDDIServiceConfig` in `ExecuteEpsilonScriptsOnDDIFile()` or as parameter `DDIPackage` in `ExecuteEpsilonScripts()`). In both cases, a DDI model is returned containing the resulting model after all scripts have been executed on the input.

Note that passing the DDI model in-memory as a Thrift data structure in `ExecuteEpsilonScripts()` is required for use cases where the tool adapter is hosted on a remote machine, because file paths can only be...
accessed on a local machine. Separating the deployment of evaluation resources is significant when considering business models involving DDI script execution “as-a-service”. For instance, cloud-based runtime DDI evaluation. Under this view, the computing power of the embedded agents is insufficient to perform complex analysis operations on a DDI at runtime. Therefore, embedded agents instead employ cloud resources to perform reasoning upon DDIs.

```
service DDIService{
    void ExportModelToDDIFile(
        1: TDDIServiceConfig ServiceConfiguration,
        2: TDDIDDIPackage DDIPackage)
        throws (1: TDDIAbstractEpsilonScriptExecutionException EpsilonScriptExecutionException),

    TDDIDDIPackage ImportDDIModel(
        1: TDDIServiceConfig ServiceConfiguration)
        throws (1: TDDIAbstractEpsilonScriptExecutionException EpsilonScriptExecutionException),

    TDDIDDIValidationResult ValidateDDI(
        1: string DdlPath,
        2: string EvalFilePath),

    TDDIDDIPackage ExecuteEpsilonScriptsOnDDIFile(
        1: TDDIServiceConfig ServiceConfiguration,
        2: bool BackupDDIFile,
        3: bool ReturnDDIPackage)
        throws (1: TDDIAbstractEpsilonScriptExecutionException EpsilonScriptExecutionException)

    TDDIDDIPackage ExecuteEpsilonScripts{
        1: TDDIDDIPackage DDIPackage,
        2: TDDIServiceConfig ServiceConfiguration,
        3: bool ExportDDIFile,
        4: bool ReturnDDIPackage)
        throws (1: TDDIAbstractEpsilonScriptExecutionException EpsilonScriptExecutionException)
}
```

Figure 6 - DDI Tool Adapter Thrift Service Interface

From the caller’s perspective, i.e. a modeling tool that wants to invoke the script execution service via Thrift, the script execution orchestration is important. Figure 7 presents a sample code snippet in C# that involves the execution of two scripts in order on a DDI file. The DDI contains a Component Fault Tree (CFT) (`DDIToBeAnalyzed.ddi`), on which first a quantitative analysis script is executed (`QuantitativeFaultTreeAnalysis.eol`). Afterwards, a validation script (Constraints located in `ValidationConstraints.evl`) is executed on the analyzed DDI (still located in `DDIToBeAnalyzed.ddi`, as the analysis script just added the results to the input model). This sample could be employed for e.g. checking whether the analyzed failure rate of a top event satisfies a target failure rate. In this example, the target failure rate can be imposed by a requirement associated with an integrity level.
Regarding DDI model file configuration, there are certain parameters needed to be set:

1. **ReadOnLoad**: This means that the Epsilon execution engine reads the referenced model file into memory. This parameter is false when the model is not loaded as the script execution begins. A typical usage of this feature is for producing new DDI models, which will be stored in the supplied file path when the script execution completes.
2. **StoreOnDisposal**: This parameter has to be set to true, if a new DDI file is created during the execution of the script. A typical use case would be that for one script, there is a source model and a target model. For the source model, one would set `ReadOnLoad=true, StoreOnDisposal=false` and for the (not yet existing) target model, one would set `ReadOnLoad=false, StoreOnDisposal=true`.

3. **ModelName**: This parameter represents an arbitrary name of the model to be used within a script to reference the model.

4. **Alias**: This parameter is like `ModelName` referring to a name of a model to be used within the script for referencing. Due to the fact that certain scripts might be reused for DDI models with different names, **Alias** offers the possibility to use a generic identifier such as `Source` within the script and map a specific `ModelName` to the used generic identifier at configuration time.

Variable passing from one script to the next is also possible, via **Export/Import Parameters**. A hypothetical use case for this feature would be to validate only those elements in the second script that changed in the execution of the first script. The knowledge of which specific elements have been analyzed is only available in the first script. The exported elements can be collected under a common alias e.g. `ChangeElements` from the first script and imported to the validation script. The **Alias** field of the exported and imported parameters follows a similar concept as described above for the model configuration, i.e. a more expressive name can be used at configuration time while still being able to bind parameters to generic variable names within the scripts.

After Epsilon scripts, input and output DDI models have been configured, the resulting Thrift parameter `TDDIServiceConfig` is packaged together with the desired execution sequence of the scripts and the Thrift service is called returning the new DDI including the analyzed component fault tree as well as potential validation flaws.

### 3.1.3.2 Epsilon via ANT

In Figure 8, a simple example of a typical sequence of model loading, validation, conditional modification and storage described with an Ant buildfile can be seen.
In Figure 8, the Ant buildfile prescribes a sequence of tasks to be undertaken. Under the assumption that either the store target is specified, or no target is specified, the buildfile will begin execution from the ‘store’ target. Given that store depends on the analyze_modify target, the latter will first be executed. This dependency chain continues up to the execution of the ‘load’ target, which depends on no other targets and will thus be executed first.

The load target uses an Epsilon facility for loading EMF models to load the subject model of the example from the local filesystem. It then assigns it a name known to the Ant context, SubjectModel. Following this step, an Epsilon Validation Language (EVL) script is executed against the SubjectModel. This is a typical step to ensure that the model conforms to assumptions necessary for the task at hand. If failures against the constraints specified in the EVL script are detected, the process stops here, as indicated by the failOnErrors attribute of the epsilon.evl task in the ‘validate’ target. The supplier of SubjectModel would then have to make appropriate corrections and attempt again. Assuming the model is correct, the analyze_modify target executes two EOL scripts. The first one extracts an element or property of interest from the SubjectModel to be modified and the second one generates a new model from what was extracted, named TargetModel. The final target stores the generated model into the local filesystem.

```
<project default="store">
  <target name="load">
    <epsilon.emf.loadModel
      name="SubjectModel"
      modelfile="subject.model"
      metamodelfile="metamodel.ecore"/>
  </target>
  <target name="validate" depends="load">
    <epsilon.evl src="check_constraints.evl"
      failOnErrors="true">
      <model ref="SubjectModel"/>
    </epsilon.evl>
  </target>
  <target name="analyze_modify" depends="validate">
    <epsilon.eol src="analyze.eol">
      <model ref="SubjectModel"/>
      <exports ref="modificationTarget" as="modificationTarget"/>
    </epsilon.eol>
    <epsilon.eol src="modify.eol">
      <uses ref="modificationTarget" as="modificationTarget"/>
      <exports ref="TargetModel" as="TargetModel"/>
    </epsilon.eol>
  </target>
  <target name="store" depends="analyze_modify">
    <epsilon.storeModel
      name="TargetModel"
      target="target.model"/>
  </target>
</project>
```
3.1.4 External Service Invocation

The DDI scripting concept based on the Epsilon Framework enables an efficient way to express DDI manipulation algorithms independent of a specific implementation within the environment of a concrete modeling tool. In theory, this means that many dependability-related algorithms and checks that manipulate or query DDIs could be expressed in an Epsilon language.

However, existing dependability modeling and analysis tools have rich feature sets, which would have to be reimplemented in Epsilon to be used on DDI models. To overcome this issue and minimize reimplementation efforts, a generic Thrift service interface has been implemented in the common tool adapter. Through this interface, algorithms of external tools can be called from within an Epsilon script and the algorithm results can be used directly in the remainder of the script. The concrete Thrift definition of the service is shown in the lower part of Figure 6. Note that this service is unlike the previously explained services (e.g. ImportDDI() or ExecuteEpsilonScripts()), as it is provided by the modeling tools and consumed by DDI tool adapter.

Figure 9 illustrates the concept for using an external tool service in an Epsilon script. In the example, an Epsilon script is used to perform a quantitative fault tree analysis on a DDI model. The DDI contains a component fault tree representation incorporating parameterized quantitative failure rates for basic events, but not yet analyzed failure rates for the top events of interest. Within the script, it is first necessary to identify the concrete fault tree as well as the concrete top event to be analyzed (in case there are numerous) within the DDI. Afterwards, the DDI model together with the analysis parameter (top event) is provided to the ExternalServiceImpl. The latter is a Thrift service client that hands over the DDI model and service identifier (“AnalyseFaultTree” in the example) to the generic Thrift service. The code snippet for invoking the service in the Epsilon script is shown in Figure 10. Note that the ExternalServiceImpl has been implemented as part of the DDI tool adapter and is usable out-of-the-box for Epsilon script developers using the tool adapter.
Having explained how to call the external service, Figure 11 shows a C# example how the generic `InvokeExternalService()` Thrift service implementation looks like in a modelling tool providing the service. When the service is being invoked in the Epsilon script, the Thrift middleware serializes the data parameters (`DDIPackage`, `ServiceId` and `ServiceParameters`) and sends them via communication layer to the listening Thrift server (i.e. the Service Handler in the modeling tool). For more information on how a Thrift service handler is implemented for a given contract definition, please refer to the official Thrift website[^4]. A fundamental advantage of Thrift in this respect is that it is target-language-agnostic, i.e. there exists support for a great variety of target programming languages that can be used for providing the service, the most widespread being C#, Java, Python and C++.

![Figure 10 - External Service Invocation via Epsilon Script](image)

An important requirement for providing external services is the freedom to define and use new ones as the need arises in the dependability engineering lifecycle. Therefore, the `ExternalService` has been designed with an aim to maximize flexibility and decouple it from the specifics of the modeling tool service. To this end, we use the `ServiceId` parameter to uniquely identify the required algorithm on the caller side (i.e. in the Epsilon script) as well as the provider side (i.e. in the service handler). With this procedure, new services can be easily added by establishing a set of “algorithm identifiers” that can be directly used as such within the script.

[^4]: [https://thrift.apache.org/](https://thrift.apache.org/)
4 Semi-Automatic DDI Case Study

4.1 ETCS

The SIEMENS implementation of the On-Board Unit and Trackside Unit of the European Train Control System (ETCS) is a central use case of the DEIS project. As it has been already been presented in previous public deliverables, the reader is referred to D5.1 and D4.2 for more background details.

Note that the activities described in this section only represent a small subset of the activities that can be performed with the DDI scripting framework. For the purpose of this document, a subset of dependability activities from the ETCS use case are presented, illustrating the technical capabilities of the DDI technology described in Section 3.

For D4.4, the use case involving the OEM and Tier-N supplier(s) is extended to demonstrate the utility of the semi-automated DDI features introduced. Specifically, the previously manually created and integrated DDIs exchanged by parties involved in the supply chain are now semi-automatically generated and composed. For the purposes of the present deliverable, the overall system is the overall ETCS system (including the On-Board Unit) and the supplied system is the Trackside Unit. An overview of this scenario can be seen in Figure 12.

![Figure 12 - Overview of ETCS with Semi-Automatic Integration](image-url)
The following steps describe the process depicted in Figure 15:

1. First, the supplier provides a DDI to the OEM with information describing architecture, dependability requirement assurance and other relevant information regarding their supplied subsystem. The OEM semi-automatically integrates the provided DDI’s safety case fragment into the ETCS safety case. The OEM also integrates the system architectures from the DDIs.

2. The integrated DDI can now be evaluated by the OEM using semi-automated tool support: For the demonstration, the OEM applies quantitative Fault Tree Analysis (FTA) to verify whether the supplied subsystem does not cause violation of the overall system’s failure rate.

3. Appropriate notification of success or failure of the test is provided as feedback, informing the user to improve the design or choice of subsystem(s). If the validation fails, the designer should revisit their choices; however, some of this effort can be supported via scripts, see section 4.1.4.

4. If the validation is successful, evidence can be automatically generated in the assurance argument based on the analysis performed.

The following sections provided a more detailed description of the above steps. Towards this end, the newly-developed DDI capabilities described in Section 3 are also utilized.

4.1.1 Semi-Automatic DDI Integration
The synthesis process of DDIs (integration of multiple DDIs) is performed by executing a number of model management programs written in the Epsilon Object Language (EOL). The following steps are required to perform DDI integration:

- Preparation of the DDIs for integration. This step investigates whether the DDIs are ready for integration. In order for AssuranceCasePackages to be integrated, they need to declare their AssuranceCasePackageInterfaces which enumerates the contents of the AssuranceCasePackages (i.e. the ArtifactPackages, ArgumentPackages and TerminologyPackages) and declare their corresponding interfaces (i.e. ArtifactPackageInterfaces, ArgumentPackageInterfaces and TerminologyPackageInterfaces). This step constructs the interfaces for the assurance case packages of the participating DDIs. Ideally, the interfaces should be provided by the DDI creators (for IP protection and security reasons), but this step is performed for both the ETCS and Trackside DDIs for demonstration purposes.

- Identifying integration points. Dependability is not a compositional property. Therefore, OEMs should identify the integration points, which explicitly specify the subtle dependencies among (the majority of) architecture/requirements/safety and/or security goals. In this step, the creators of the DDI need to identify the integration points and place them in the corresponding interfaces mentioned before. Ideally, this step should also be performed by the DDI creators, but this step is performed for both the ETCS and Trackside DDIs for demonstration purposes.

- With the integration points defined in the interfaces of each of the DDIs, this step performs the integration. For assurance cases, an AssuranceCasePackageBinding is created, which enlist the afore created AssuranceCasePackageInterfaces in them to indicate the composition of the
assurance cases. In turn, ArgumentPackageBindings, ArtifactPackageBindings and TerminologyPackageBindings are created to “bind” the interfaces created.

Figure 13 - SACM Integration EOL Script

The integration of the remaining packages in the DDI follows the same pattern, as the packaging mechanism of the ODE was designed in a similar fashion to SACM. The OEM must identify integration points and provide mappings from the system interface to the subsystem interface. Figure 14 presents the EOL script operation enabling such architecture integration.

Figure 14 - DesignPackage Integration EOL Script

The script operates as follows:
1. The operation receives as input the system’s name (‘ETCS’ in the example), the subsystem’s name (‘Trackside’ in the example) as well as two Maps. The inputPortMapping maps system input ports to subsystem output ports and vice versa for the outputPortMapping (line 3).

2. The operation begins by retrieving the system from the OEM’s own DDI model and the DesignPackage containing it (lines 5-6).

3. The subsystem is retrieved from the supplier’s DDI (line 8).

4. A deep copy of the supplier’s subsystem is required if the OEM wishes to maintain the DDI as a single file. Note that this is a user-defined operation (Clone) that invokes platform-dependent copy operations on the subsystem and its dependencies (FailureLogicPackages and DependabilityPackages). The operation also adds the new subsystem under the system’s DesignPackage, its associated FailureLogicPackage and DependabilityPackage under the OEM DDI’s DDIPackage (line 10).

5. The system adds a reference to the subsystem (line 12).

6. The mappings from input to output ports are iterated through; new Signals linking input ports to output ports are added to the DesignPackage (lines 14-20 and 22-28).

Due to the dependency of the DesignPackage on the FailureLogicPackage, linking the ETSC system to the Trackside system is sufficient. Provided the Clone operation mentioned above is correctly implemented, the references between the other ODEProductPackages imported should be maintained.

4.1.2 DDI Dependability Analysis via External Service Requisition

Having integrated the assurance case fragment of the Trackside unit supplier into the ETCS assurance case automatically, one task that needs to be carried out by the dependability engineers of the ETCS system is the verification of whether tolerable failure rates have been achieved through the system design. One way to verify a failure rate demand such as “Hazard rate of hazardous event ‘Erroneous Eurobalise telegram/telegrams interpretable as correct’ <= 1e-11/h” (G11 in Figure 15) is to perform a quantitative analysis on a (component) fault tree that has been modeled for the hazardous event of the Trackside Unit.

Within the ETCS System DDI, G11 represents the target failure rate requirement that needs to be satisfied by the Trackside Unit. In the Trackside Unit DDI, all models required for checking the validity of this requirement are existent. In order that the ETCS dependability engineer can perform safety analysis across integration borders, goal G11 must be enriched with reference information:

a) Which model elements should be subject to requirement checking (e.g. the top event “Erroneous Eurobalise telegram/telegrams interpretable as correct” in component fault tree “ETCS Trackside.cft”)

b) What exactly needs to be done with those model elements (e.g. perform a quantitative fault tree analysis). Technically, this information is added to the SACM Claim via TaggedValues. In Figure 15, G11 contains two tagged values:

- **Activity=“QuantitativeFTA”:** The action that should be executed via DDI script is a quantitative fault tree analysis
• **TopEvent=<modelElementIdOfTopEvent>**: The unique DDI ID of the DDI element representing the top event to be analyzed. Note that this ID is set in the Trackside Unit’s assurance case fragment, which also contains G11 as a top-level claim.

Note that SACM tagged values are just the first step to enable referencing elements. In upcoming versions of the ODE, the *SACM Terminology package* contains mechanisms for enriching prose text with formal content as in G11 by means of structured text, expressed as *Terms* and *Expressions*.

![Figure 15 - Referencing Elements in Different DDIs](image)

To verify that the Trackside CFT top event failure rate satisfies demand, the integrator side’s dependability engineer must perform quantitative analysis. A plausible approach to automate this task would be to implement an Epsilon script. The script locates the top event in the CFT and performs a quantitative analysis, using the structure of the Trackside CFT.

The Epsilon Object Language (EOL) is capable of expressing such algorithms. However, for demonstration purposes, we can assume that the ETCS integrator already has a modeling and analysis tool deployed capable of performing the required quantitative FTA (such as safeTbox, HiP-HOPS or ComposR). Thus, the OEM opts to use the existing tool’s analysis capabilities from within the Epsilon script.

Figure 16 presents the Epsilon EOL script capable of performing this activity according to the concept described in Section 3.1.4. The goal of this script is to perform a quantitative analysis on a top event that is referenced in a SACM Claim by means of an external service invocation. After analysis, the input DDI is updated with the quantitative analysis results and ready to compare the target and actual failure rates (see next subsection). Lines 13-40 query the DDI to find the fault tree and top event to be provided to the external analysis service. Lines 42-56 invoke the external analysis service. Note that Lines 42-56 are equal to the snippet shown in Figure 10.
4.1.3 Verification of Process & Product-Related Dependability Requirements

The previous subsection described how to analyze parts of the DDI model and thereby enrich the DDI with new information (e.g., analysis results such as FTA-based failure rates). The next step in the process is to confirm whether target requirements are met or whether additional dependability mechanisms must be employed. This task can be fully automated using DDI scripts.

Dependability requirements typically refer to one of two categories:

- they are related to the quality of the (dependability) engineering process to be performed.
- they demand a concrete property to be met by the developed product.

Ideally, in a proper dependability engineering process, both types of requirements are explicitly documented as part of the assurance case. A sample assurance argument for the ETCS system is shown in Figure 15, on the left side. In the example, Claim G14 is a process requirement demanding the execution of processes recommended for a SIL 4 rated system. Claim G11 is a product requirement and demands the failure rate of a certain hazardous event to not exceed a target failure rate.
Figure 17 - Involved DDI Elements for Checking Failure Rate Satisfaction Requirement

Figure 17 shows the DDI elements involved in analyzing the satisfaction of a failure rate requirement. As described above, the SACM Claim G11 expresses the requirement itself. Claim G11 contains additional data elements that allow the DDI script to automatically check the satisfaction of the requirement.

1. The reference to the TopEvent element in the Trackside CFT representing the hazardous event described in the requirement.
2. The TargetFailureRate to be satisfied by the actual computed failure rate for the fault tree (assumed to be existent in the failureRate attribute of the top event’s ODE representation). Note that top events are modeled as Gates in the ODE. Through this abstraction, top events represent elements that can have both incoming and outgoing propagations, just like Gates.

In order to verify ETCS dependability requirements with the DDI technology, the Epsilon Validation Language (EVL) can be used to check structural constraints on a given model. The verification activity itself is assumed to be a knowledge-gathering task, i.e. it only reviews existing data and reasons about it to produce new knowledge, vis-à-vis whether a requirement been met. Therefore, the verification activity has been decoupled from the synthesis of the data required for reasoning (e.g. the analysis results).
```javascript
// This script checks if the top analysed top event, which is associated by a claim, 
// holds a failure rate that is less or equal to the failure rate required by the claim.

context DDIModel!Claim {
    constraint associatedTopEventsFailureRateFulfillsRequirement {
        guard: (self.taggedValue.exists(tv | tv.key.value.exists(keyLangString | keyLangString.content = "TopEvent")) and
                 self.taggedValue.exists(tv | tv.key.value.exists(keyLangString | keyLangString.content = "TargetFailureRate")))
        check {
            var fulfillsReq = false;
            var associatedTopEvent = self.getAssociatedTopEventElement();
            if(associatedTopEvent != null) {
                var failureRateTopEvent = associatedTopEvent.failure.failureRate;
                if(associatedTopEvent.failure.isDefined and associatedTopEvent.failure.failureRate > 0.0) {
                    var requiredFailureRateAsDouble = self.getRequiredFailureRate();
                    if(requiredFailureRateAsDouble != null and requiredFailureRateAsDouble > 0.0) {
                        fulfillsReq = failureRateTopEvent <= requiredFailureRateAsDouble;
                    }
                }
            }
            return fulfillsReq;
        }
    }
}

operation DDIModel!Claim.getAssociatedTopEventElement(DDIModel!Gate) {
    var topEvent : DDIModel!Gate;
    var topEventModelElementId = "";
    var topEventIdTaggedValue = self.taggedValue.select(tv | tv.key.value.exists(keyLangString | keyLangString.content = "TopEvent")).first();
    if(topEventIdTaggedValue != null) {
        var topEventModelIdToString = topEventIdTaggedValue.content.value.first();
        var topEventModelId = topEventModelIdToString;
    }
    if(topEventModelId != "") {
        var elementById = DDIModel!.getElementById(topEventModelId + "", "Gate") and elementById.type == DDIModel!.GateType.outputEvent{
            topEvent = elementById;
        }
    }

    return topEvent;
}

operation DDIModel!Claim.getRequiredFailureRate(DDIModel!Gate) {
    var targetFailureRateAsString = "";
    var targetFailureRateAsDouble;
    var targetFailureRateTaggedValue = self.taggedValue.select(tv | tv.key.value.exists(keyLangString | keyLangString.content = "TargetFailureRate")).first();
    if(targetFailureRateTaggedValue != null) {
        var targetFailureRateAsString = targetFailureRateTaggedValue.content.value.first();
        if(targetFailureRateAsString != "") {
            targetFailureRateAsDouble = convertFromStringToDouble(targetFailureRateAsString);
        }
    }
    return targetFailureRateAsDouble;
}
```

Figure 18 - EVL Script for Validating Quantitative Analysis Requirements
For the ETCS example, this means, a DDI script for checking failure rate satisfaction (see Figure 18) has to perform three tasks:

1. Retrieve those claims in the assurance argument representing quantitative analysis requirements (indicated by a TargetFailureRate tagged value)
2. For each identified claim, locate the referenced top event element in the FailureLogicPackage DDI and retrieve the actual failureRate attribute
3. Compare actual failure rate with target failure rate and report a violation to the dependability engineer, if the target failure rate is smaller than the actual failure rate

Note that in the case of the concrete EVL script above, constraints are specified for a certain element type (such as Claim). The benefit of using EVL in this scenario is that the traversal of the DDI and checking the specified constraint on each Claim is performed by the Epsilon framework automatically. Thus, the dependability engineer can focus on the definition of the problem solution (i.e. what needs to be really checked) instead of writing boilerplate code for DDI traversal.

For process-related requirements, the technical procedure for requirement verification is similar to the procedure for product-related requirements, i.e. the Epsilon Validation Language (EVL) is used to check constraints on attributes of elements or the existence of certain kinds of elements and relations. Figure 19 shows an excerpt from Trackside Unit’s DDI of the ETCS system.

The requirements that should be automatically checked for satisfaction by DDI scripts are expressed in claims G16: “FMECA is performed and its results have been documented” and G17: “Independent reviews of FMECA have been performed by two persons.”. As indicated in the figure, each claim is related to models representing the evidence for the claim via ArtifactReferences. For both claims G16 and G17, the evidence can be checked by looking at the contents of the ArtifactPackage containing information about the processes that have been carried out with respect to the Artifact of interest “Hazard and Failure Analysis Result”.

In order to automatically check the satisfaction of G16 and G17, a DDI script in EVL needs to codify the following conditions:

- **G16**: The Artifact related to the ArtifactReference related to the Claim of interest should have an associated Activity “Hazard or Failure analysis”, where the used Technique is “FMECA”.
- **G17**: The Artifact related to the ArtifactReference related to the Claim of interest should have two associated Activities “Review”, where the associated Participants for each review are from different companies.

Note that processes should be standardized (at least for a given application context) and thus the terminology used in requirements expressed as Claims in SACM and the evidence located in an ArtifactPackage needs to match. The DEIS consortium is currently analyzing the results of the H2020 AMASS
project\textsuperscript{5} to see how the process terminology can be expressed in a dedicated package and reused in \textit{ArgumentPackages} and \textit{ArtifactPackages}.

![Diagram of SACM Argument Package and SACM Artifact Package]

\textbf{Figure 19 - ETCS Process Requirements and Model Elements Containing Evidence}

4.1.4 Automatic Generation of Dependability Evidence based on DDI Analysis Results

Once the EVL validation has been performed, there are two possibilities:

a) The first is the case where validation is successful. In this scenario, the integrator can proceed to complete the integrated system’s assurance case by attaching the quantitative FTA results as evidence under the relevant claims.

b) In the alternative scenario, validation is unsuccessful. This would mean that the quantitative FTA results indicate that a TargetFailureRate (attached to a given Claim) has been violated. In this

\textsuperscript{5} \url{https://www.amass-ecsel.eu/}
scenario, the integrator must choose to either alter the system design (e.g. introduce redundancy) to reduce the offending TargetFailureRate or employ more rigorous development assurance processes to mitigate the potential hazards.

In the first scenario, the dependability engineer can setup conditional execution of an EOL script, provided the validation is successful. The script then uses the updatedDDI received from the external tool invocation, as seen in Figure 16 (line 51) to generate new elements for the assurance case. Specifically, the results of the quantitative FTA are represented by a new Artifact entry in the ArtifactPackage of the AssuranceCasePackage of the DDI. To support the relevant claims G11 and G12 in the ArgumentPackage (representing the claims depicted in Figure 15), ArtifactReferences linked to the newly created Artifact are introduced to the ArgumentPackage. The ArtifactReferences are linked via AssertedEvidence relationships to the above claims. Once this process is complete, the dependability assurance case is now fully supported with concrete evidence and can be reviewed. Traceability from the assurance case to the supporting material is also maintained, as the results of the FTA, alongside the remaining supporting material, is stored together within the integrated DDI.

The script seen in Figure 20 demonstrates how this process can be programmed in EOL. The script defines an operation, a named block of statements, which accepts 3 inputs as parameters. The first is the TopEvent, representing the DDI-unique identifier of the top event of the analyzed fault tree. The subjectClaim is the name of the claim being supported with the analysis results, so this would include ‘G11’ and ‘G12’. Finally, the solutionNum is a number used to differentiate the title of the newly generated solution elements in the assurance case.

```java
operation GenerateQFTAEvidence(TopEvent : String, subjectClaim : String, solutionNum : String) { 
    var topEvent = updatedDDI.getElementById(TopEvent);
    if(not topEvent.isDefined())
        return;
    var artifactName = “QFTA Ext:” + topEvent.name;
    var artifactDescription = “Quantitative fault tree analysis of the Top Events “ + “\" + topEvent.name + “\";
    var qftaResultArtifact = updatedDDI.ArtifactPackage.all().first().BuildArtifact(artifactName, artifactDescription);
    var claim = updatedDDI!ArgumentPackage.all().first().argumentationElement.select(e | e.GetName() = subjectClaim);
    if(not claim.isDefined())
        return;
    var evidenceTitle = “Sn” + solutionNum;
    var evidenceDescription = artifactDescription + “ interpretable as correct”;
    var solution = updatedDDI!ArgumentPackage.all().first().BuildSolution(evidenceTitle, evidenceDescription, qftaResultArtifact);
    LinkGoalToSolution(claim, solution);
}
```

Figure 20 - Evidence Generation Script

Given the above inputs, the script in Figure 20:

1. Retrieves the fault tree’s top event element stored within the DDI via its identifier, TopEvent (line 4)
2. Generates a new SACM Artifact element describing the fault tree results (lines 9-11)
3. Retrieves the SACM Claim corresponding to G11 or G12 from Figure 15 (line 13)
4. Generates a new SACM ArtifactReference linked to the new Artifact element, corresponding to Sn5 or Sn7 from Figure 15 (lines 18-20)
5. Links the generated ArtifactReference to the subject claim (G11/G12) (line 22)

If the validation fails, then the OEM needs to find a way of satisfying the violated requirement. There is no definitive approach here; for demonstration purposes, the OEM is assumed to have decided to replace the offending subsystem with a more expensive but also more reliable one. To do so, when the result of the validation notifies a failed check against the TargetFailureRate, the OEM has configured a conditional EOL script to be executed. The script can be seen in Figure 21.

```
operation DeployReliableSubsystem(subjectClaim : String, TopEvent : String) {
    var claim = updatedDDI!ArgumentPackage.all().first().select(g | g.GetName() = subjectClaim);
    var taggedValue = claim.taggedValue.selectOne(y | y.GetKey() = "AlternativeSubsystemId");
    var alternativeSubSystemId = taggedValue.content.value.first().content;
    var alternativeSubSystem = updatedDDI.getElementById(alternativeSubSystemId);
    var topEvent = updatedDDI.getElementById(TopEvent);
    var allPorts = updatedDDI!Ports.all();
    var portContainingTopEvent = allPorts.selectOne(p | p.interfaceFailures.includes(topEvent.failure));
    var allSystems = updatedDDI!System.all();
    var owningSystem = allSystems.selectOne(s | s.ports.includes(portContainingTopEvent));
    ReplaceSystem(owningSystem, alternativeSubSystem);
}
```

**Figure 21 - EOL Script for Alternative Subsystem Replacement**

In the script:

1. The subject claim being violated (G11 or G12 in Figure 15) is retrieved (line 4)
2. The OEM has configured an alternative subsystem in the DDI. Furthermore, the alternative’s identifier has been stored under one of the claim’s TaggedValues, with the key being “AlternativeSubsystemId”. The script retrieves the alternative subsystem from within the DDI (lines 6-8)
3. The top event of the fault tree which was evaluated previously is retrieved as per Figure 20 (line 10)
4. The system owning the top event is identified and retrieved (lines 12 – 16)
5. The system owning the top event is replaced by the alternative subsystem (line 18)

The user-defined operation invoked on line 18 is omitted due to space limitations; it simply finds and replaces references to the existing subsystem with references to the alternative.

Once the script has been executed, quantitative FTA and validation can be reapplied, as indicated in Figure 12. Provided the results of the FTA and the validation are satisfactory, the procedure returns to the first scenario mentioned in this section.
5 Further Discussion and Conclusion

The DDI has already been established in previous work as a vehicle useful for connecting dependability assessment and assurance methodologies, the latter being implemented across a heterogeneous tool ecosystem. And while the utility of being able to transfer information seamlessly across toolchains cannot be understated, the development and operation of open and adaptive Cyber-Physical Systems poses further challenges. To address some of these challenges, the DEIS project is investigating the semi-automatic synthesis, integration and evaluation of DDIs.

In this deliverable, the current progress of the DEIS consortium towards enabling semi-automatic DDI features has been presented. Programmable control of DDI structure and content using a family of script-based languages, Epsilon, has been introduced. Management and execution scripts have also been addressed via two avenues, the Ant build system and the DEIS Common Tool Adapter. Ant is an established solution with widespread usage in typical development environments. However, adopting the Ant approach necessitates additional effort to fully support DDI import/export and external tool invocation from scripts.

The DEIS common tool adapter largely addresses these concerns. Interested parties can connect their tools directly via the adapter’s Application Program Interface (API). The adapter functions not only just as a bridge between tool APIs but also between their respective programming languages. Using the Thrift middleware library, the domain of programming languages supported via this API is extensive, featuring more than 25 languages⁶. This includes major general-purpose programming languages such as Java, C/C++ and C#. Finally, Epsilon script orchestration and execution, as well as external tool invocation are immediately available via the API.

To demonstrate the feature set introduced, the SIEMENS use case for the European Train Control System (ETCS) is extended from D4.2. The scenario between an OEM and Tier-N supplier is revisited, with the OEM using semi-automatic tool support to alleviate effort in integration, validation and assurance case fragment generation. Specifically, the OEM, upon receipt of a DDI describing a supplier’s subsystem, can initiate a chain of actions upon the received and the host system’s DDIs.

In the DEIS project proposal, the present deliverable also included demonstration of semi-automatic synthesis based on requirements from the CENELEC EN and ISO 26262 standards. In the use case presented for the current deliverable, for the SIEMENS implementation of the ETCS, only the CENELEC EN standards are applicable. However, the General Motors Dependable Physiological System (DPMS) use case involves ISO 26262 requirements and is actively investigated. The knowledge and progress made towards supporting the ETCS use case with semi-automatic DDIs will be transferred to the DPMS use case over the course of the project. In doing so, application of the semi-automatic features for synthesizing DDIs based on ISO 26262 requirements will also be explored.

⁶ https://thrift.apache.org/docs/Languages
It should be noted that the work so far has been prototypical, for feasibility demonstration purposes. For applicability in more realistic scenarios further work needs to be done:

- DDI integration using task specific languages. Currently, the Epsilon Object Language is used for the integration process. In the future, the capabilities offered by Epsilon Merging Language (EML) will be investigated. If DDI integration is not practical via EML, it is also possible to create a new language by extending EOL syntax for this specific task.

- Automated integration preparation. To prepare the DDIs for integration, as previously mentioned, interfaces need to be created. For future work, the creators of the DDIs need only specify the integration points, the interfaces for DDI integration would be automatically generated (similar fashion to the integration process demonstrated before).

- Automated generation of the integration programs. To enable automation at runtime, in the future it is desirable to have a service which generates the integration programs automatically, based on the integration points provided by DDI creators at design time. With the integration points specified by DDI developers, it is possible to generate programs written in Epsilon languages which provide the integration functions.

- Extended coverage of dependability requirements via script support. As mentioned previously, a larger set of applicable standards, starting with ISO 26262, can be supported. Furthermore, use of assurance case patterns [9] can provide more expressive power and flexibility to assurance case designers. Thus, implementing SACM pattern support would be another significant feature towards DDI semi-automated synthesis.

As the progress on DEIS continues, the consortium partners are actively further developing the available tool base to transition from design-time to runtime DDIs. This step will require extending DDI synthesis, integration and evaluation to be applicable for in-the-field CPS.

6 References


