Advancing Traffic Efficiency and Safety through Software Technology phase 2 (ATESST2)

<table>
<thead>
<tr>
<th>Report type</th>
<th>Deliverable D3.2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Report name</td>
<td>Refined EAST-ADL2 tool support</td>
</tr>
<tr>
<td>Dissemination level</td>
<td>PU</td>
</tr>
<tr>
<td>Status</td>
<td>Final</td>
</tr>
<tr>
<td>Version number</td>
<td>1.0</td>
</tr>
</tbody>
</table>
### Authors

<table>
<thead>
<tr>
<th>Editor</th>
<th>E-mail</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rolf Johansson</td>
<td><a href="mailto:rolf.johansson@mentor.com">rolf.johansson@mentor.com</a></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Authors</th>
<th>E-mail</th>
</tr>
</thead>
<tbody>
<tr>
<td>Matthias Biehl</td>
<td><a href="mailto:biehl@md.kth.se">biehl@md.kth.se</a></td>
</tr>
<tr>
<td>DeJiu Chen</td>
<td><a href="mailto:chen@md.kth.se">chen@md.kth.se</a></td>
</tr>
<tr>
<td>Helko Glathe</td>
<td><a href="mailto:Helko.glathe@carmeq.com">Helko.glathe@carmeq.com</a></td>
</tr>
<tr>
<td>Henrik Lönn</td>
<td><a href="mailto:henrik.lonn@volvo.com">henrik.lonn@volvo.com</a></td>
</tr>
<tr>
<td>Yiannis Papadopoulos</td>
<td><a href="mailto:y.i.papadopoulos@hull.ac.uk">y.i.papadopoulos@hull.ac.uk</a></td>
</tr>
<tr>
<td>Mark-Oliver Reiser</td>
<td><a href="mailto:mark-oliver.reiser@tu-berlin.de">mark-oliver.reiser@tu-berlin.de</a></td>
</tr>
<tr>
<td>David Servat</td>
<td><a href="mailto:david.servat@cea.fr">david.servat@cea.fr</a></td>
</tr>
<tr>
<td>Martin Walker</td>
<td><a href="mailto:martin.walker@hull.ac.uk">martin.walker@hull.ac.uk</a></td>
</tr>
<tr>
<td>Carl-Johan Sjöstedt</td>
<td><a href="mailto:carlj@md.kth.se">carlj@md.kth.se</a></td>
</tr>
</tbody>
</table>

### The Consortium

<table>
<thead>
<tr>
<th>Volvo Technology Corporation (S)</th>
<th>VW/Carmeq (D)</th>
<th>Centro Ricerche Fiat (I)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Continental Automotive (D)</td>
<td>Delphi/Mecel (S)</td>
<td></td>
</tr>
<tr>
<td>Mentor Graphics Hungary (H)</td>
<td>CEA LIST (F)</td>
<td></td>
</tr>
<tr>
<td>Kungliga Tekniska Högskolan (S)</td>
<td>Technische Universität Berlin (D)</td>
<td>University of Hull (GB)</td>
</tr>
</tbody>
</table>
## Revision chart and history log

<table>
<thead>
<tr>
<th>Version</th>
<th>Date</th>
<th>Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>2010-05-03</td>
<td>First release based on existing information</td>
</tr>
</tbody>
</table>
# Table of contents

Authors ...............................................................................................................................................................2

Revision chart and history log ................................................................................................................................3

Table of contents ................................................................................................................................................4

1 Introduction .................................................................................................................................................6

2 Tool Support for Requirements ..................................................................................................................7

  2.1 Use Case Description of EAST-ADL2 RIF Plug-in .............................................................................7

    2.1.1 Use Case 1 – Instantiate new EAST-ADL2 Model based on RIF File .......................................7

    2.1.2 Use Case 2 – Import RIF File Data into preexisting EAST-ADL2 Model ...................................8

    2.1.3 Use Case 3 – Export a certain part of the Requirements Specification out of an already existing EAST-ADL2 Model ..................................................10

  2.2 Realization of the EAST-ADL2 RIF Plug-in ......................................................................................12

    2.2.1 Initial Situation ..........................................................................................................................12

    2.2.2 Architecture ..................................................................................................................................14

    2.2.3 Detailed Realization ..................................................................................................................14

  2.3 Known Limitations ............................................................................................................................17

3 Tool Support for Dependability .................................................................................................................19

  3.1 EAST-ADL2 Dependability Analysis Plugin ......................................................................................19

    3.1.1 Tool Integration ..........................................................................................................................19

    3.1.2 Semantic Mapping Transformation ..........................................................................................21

    3.1.3 Representation Transformation ................................................................................................22

  3.2 Extensions to the HiP-HOPS Dependability Analysis Engine ..........................................................24

    3.2.1 Harmonisation of EAST-ADL2 and HiP-HOPS ........................................................................24

    3.2.2 Extensions to HiP-HOPS ..........................................................................................................25

4 Tool Support for Variability .......................................................................................................................33

  4.1 CVM Framework Usage ...................................................................................................................33

5 Tool Support for Behavior modelling ........................................................................................................44

  5.1 Engineering scenarios ......................................................................................................................44

  5.2 Modelling concepts ..........................................................................................................................45

    5.2.1 Language aspects ....................................................................................................................45

    5.2.2 Basic models-of-computation in automotive embedded systems ............................................45

    5.2.3 Integration of models ................................................................................................................46

  5.3 Methodology .....................................................................................................................................47

  5.4 Plugin ................................................................................................................................................48

  5.5 Conclusion/Discussion ....................................................................................................................... Error! Bookmark not defined.

6 References ...............................................................................................................................................51

© 2010 The ATESTST2 Consortium
1 Introduction

This document is a summary of the tool support for the EAST-ADL2 language. The actual plugins and their documentation are part of the modelling and analysis workbench, D4.2.1 and D4.3.1. This deliverable provides a description of the background and purpose of the EAST-ADL tool support developed in each work package.
2 Tool Support for Requirements

It is typical for the automotive domain that the engineering of embedded systems is done across several stakeholders, for example a vehicle manufacturer and tier-1 supplier may share the development effort of a certain functionality. One typical use case is that a manufacturer setup an initial requirements specification, share certain parts of it with several suppliers to adjust the specification according to changes or comments made by the suppliers. That's why the exchange of requirements specifications is not only unidirectional but also a bidirectional flow of data (in a round-trip manner). During such round-trip exchanges, requirements and also the structuring of requirements may be changed on manufacturer or supplier side and the receiver of such a changed specification (manufacturer or supplier) may adjust its current specification according to the changes made before.

To support the exchange of requirements specifications across several partners, the german automotive organization *Herstellerinitiative Software* (**HIS**) started a definition of a tool independent representation of such specifications [http://www.automotive-his.de/rif]. The outcome of this definition is the XML based *Requirements Interchange Format* (**RIF**) which is specified as an XML Schema (**XSD**) that is including all the concepts for representing requirements specifications including the structuring of requirements and relations between them. So, using RIF for exchanging requirements specifications, tools such as IBM Rational DOORS [www.ibm.com/software/awdtools/doors], Polarion Requirements [www.polarion.com/products/requirements], Borland Caliber RM [www.borland.com/de/products/caliber/rm.html] and the EAST-ADL2 tool suite should support this exchange format. To keep track of certain requirement elements exchanged across several partners (remember the round trip manner mentioned above), RIF gives all its elements a globally unique identifier (**UUID**, [http://tools.ietf.org/html/rfc4122]).

Thanks to the EAST-ADL2 RIF plug-in, the EAST-ADL2 tool suite supports three use cases for exchanging requirements specifications via RIF:

1. Instantiate a new EAST-ADL2 model based on a given RIF file. The new model will then contain an initial requirements specification.
2. Import a requirements specification of a given RIF file into an already existing EAST-ADL2 model. Here, after the import has been done, requirements specification elements may be newly created, changed or deleted when comparing the already existing requirements data with the imported data.
3. Export a requirements specification out of an already existing EAST-ADL2 model into a newly created RIF file.

Section 2.1 gives a detailed description about the supported use cases. Section 2.2 describes the solution of the EAST-ADL2 RIF plug-in. Finally, section 2.3 list all the limitations that hold for the current version of the plug-in.

2.1 Use Case Description of EAST-ADL2 RIF Plug-in

In this section we are describing the three use cases in more detail including rough explanations about the conceptual and technical solution. A detailed description about the solution will be given in the following section (see section 2.2).

2.1.1 Use Case 1 – Instantiate new EAST-ADL2 Model based on RIF File
Figure 1 Instantiate new EAST-ADL2 Model with Data from given RIF File

Figure 1 depicts the principle idea behind the first use case. A RIF file containing a requirements specification exported by a third party requirements engineering tool or by the EAST-ADL2 tooling as well is representing the initial input for a new EAST-ADL2 model. Thus, on the left side of Figure 1 you can see the two gears which are representing the EAST-ADL2 RIF plug-in of the EAST-ADL2 tooling suite. This plug-in offers the possibility to derive a new EAST-ADL2 model file from a given RIF file.

As a preparation for putting in the requirements data, the root element of the EAST-ADL2 language will be instantiated and also an instance of the part where all requirements data will be located (that is the internal "Requirements Model", see metaclass RequirementsModel in Domain Model) inside an EAST-ADL2 model in general.

At the beginning of this action, the EAST-ADL2 RIF plug-in parses and validates the content of the RIF file. If this succeeds, all RIF typed elements will be transformed into appropriate EAST-ADL2 model elements using MOF based model-to-model transformation. Finally, a new EAST-ADL2 model containing all the EAST-ADL2 typed data of the original RIF file inside the "Requirements Model" has been created and will be shown within the EAST-ADL2 model viewer of the tooling suite.

2.1.2 Use Case 2 – Import RIF File Data into preexisting EAST-ADL2 Model
Figure 2 Import Data from given RIF File into a preexisting EAST-ADL2 Model

Figure 2 depicts the principle idea behind the second use case. It is slightly different to the former use case. Here, we also have a RIF file as input but on the receiving side of the requirements specification, there is also an already existing EAST-ADL2 model. Thus we do not simply instantiate a new model for the RIF file (as described for the first use case). Instead, the requirements specification of the RIF file will be imported into the already existing EAST-ADL2 model.

The data will not simply be transformed and added to already existing requirements data. For such a case, the EAST-ADL2 language provides the possibility to add an extra allocated area (symbolized by red rectangle in Figure 2) into the preexisting EAST-ADL2 model (see metaclass RIFImportArea in Domain Model). Into such an area all transformed requirements data will be put in (symbolized through red lines in red rectangle). Through the RIF Import Area, we avoid the risk of getting inconsistencies in the requirements specification after RIF data have been transformed into the exiting EAST-ADL2 model.

Finally, for all requirements elements inside the new RIF Import Area which have an identifier which is exactly the same as of an element in the Requirements Model of the EAST-ADL2 model, special references will be established (see metaclass MultiLevelReference in Domain Model) between elements with the same identifier. Those references are symbolized as dark grey arrows between requirement data elements of the red and the black box in Figure 2. Two elements, which are staying in reference through such a Multi-Level reference may have different content, but the same identifier in general. Thus some kind of resolving possible data differences should be provided. Currently there is no support for this, so the end user must resolve such differences manually.

Figure 3 shows an abstract example of a RIF Import Area and the Multi-Level references (dashed yellow arrows).
Figure 3 Abstract Example of a RIF Import Area

In this example, the current requirements model of the EAST-ADL2 model contains three requirements, where Req 1.1 and Req 1.3 are sub requirements of Req 1. Moreover, we can see a RIF Import Area which contains five requirements. Req 1’, Req 1.1’ and Req 1.3’ are symbolizing imported requirements for which requirements in the Requirements Model already exist. Thus for example, Req 1 has the same global unique identifier as Req 1’. Moreover, Req 1’, Req 1.1’ and Req 1.3’ might have slight changes in compare to their counterparts in the Requirements Model. The Multi-Level references are stating that there are requirements with identical global unique identifiers in RIF Import Areas. Req 1.2 and Req 1.3.1 do not have counterparts in the Requirements Model. Thus, no Multi-Level references have been instantiated to point on them. Req 1.2 can be seen as to be a new sub requirement of Req 1 in the Requirements Model. Req 1.3.1 can also be seen as a new sub requirement, but the parent would be Req 1.3 of the Requirements Model.

2.1.3 Use Case 3 – Export a certain part of the Requirements Specification out of an already existing EAST-ADL2 Model
Figure 4 Export of a certain Part of a Requirements Specification of an already existing EAST-ADL2 Model

Figure 4 depicts the principle idea behind the third use case. This use case is analogous to the second use case, but goes in the reverse direction. Thus, an already existing EAST-ADL2 model which is containing a requirements specification (symbolized by black rectangle in Figure 3) is now the starting point of view. The EAST-ADL2 RIF plug-in offers an action to export a certain part of the requirements specification. That means, that top level structuring elements of type RequirementContainer (elements of type RequirementContainer will be used for structuring all the requirements into a tree based structure like in 3rd party tools such as IBM Rational DOORS) will be displayed first. The end user chose one of those containers. Next, all requirements elements below the chosen container and the container itself will be cloned into a newly established extra allocated RIF Export Area (see RIFExportArea in the Domain Model). This RIF Export Area is symbolized by the blue rectangle in Figure 3. After this, Multi-Level references as described in the second use case will be established, but now those Multi-Level references are pointing from the elements inside the RIF Export Area to their original elements (see dark greyed arrows between requirement elements of black and blue rectangles in Figure 3). Thus, the end user has the possibility to keep track of changes or deletions done in the Requirements Model (in the meantime, after the clones has been produced for the RIF Export Area) in compare to the exported data of a RIF Export Area. The criterion for establishing those Multi-Level references holds as described in the second use case above (matching global unique identifiers).

Figure 5 shows an abstract example of a RIF Export Area including Multi-Level references between exported requirement elements to their origins. Only Req 1.1 does not have a Multi-Level reference to a requirement in the Requirements Model. This means, that the origin of Req 1.1 must have been deleted after the export has been done.
Figure 5 Abstract Example of RIF Export Area

Thus, the content of elements inside a RIF Export Area and their counterparts in the Requirements Model might become different over time.

Finally, after the RIF Export Area has been created and Multi-Level references has been established, the EAST-ADL2 RIF plug-in is transforming all elements of the new RIF Export Area into appropriate RIF typed elements. Thus we have finally a new RIF model stored in a RIF conform XML file which conforms to the provided RIF XSD schema.

2.2 Realization of the EAST-ADL2 RIF Plug-in

In this section we are providing a detailed description on how the EAST-ADL2 plug-in has been realized. We are describing the initial situation from a technical point of view, the architecture of the plug-in and a detailed description about the components which are responsible for the import resp. export including data flows.

2.2.1 Initial Situation

The Papyrus EAST-ADL2 modeling tool is based on the Eclipse Framework. Thus the EAST-ADL2 RIF plug-in shall be developed as an Eclipse plug-in which extends the EAST-ADL2 Eclipse Tool Suite for giving the user the possibility to import resp. export requirements specifications into and out of an EAST-ADL2 model.

The requirements interchange data will be based on RIF. The RIF specification has been developed as a XML schema. Thus, RIF data will be interchanged as XML files based on this schema.

The EAST-ADL2 language has been developed as an UML Profile based on the Eclipse EMF UML Extension.

The transformation of data from RIF to EAST-ADL2 resp. from EAST-ADL2 to RIF shall be developed as a model-to-model transformation using the ATL transformation engine (ATL is also provided as an Eclipse Extension). An ATL transformation requires the metamodel information of the source model and the metamodel information of the target model. Both metamodels must be based on the EMF Ecore meta-metamodel. The input and the target model must be provided as EMF Models based on their EMF metamodels. The EAST-ADL2 UML profile, which is one of the metamodels for the ATL transformation engine, already fulfills this requirement, because it has been developed with the Eclipse EMF UML Extension. Thus it is implicitly an EMF based metamodel. Instead, the RIF XML schema does not fulfill this requirement, but thanks to the capability of the Eclipse EMF Modeling Framework, the RIF XML schema can be transformed into...
an appropriate EMF Ecore model using the XML schema transformation functionality of the framework. Thus, we have derived the EMF Ecore based RIF metamodel by using this feature. Moreover, the EMF tooling also provides mechanisms for serializing and deserializing XML models based on their XML schemas which must be transformed into EMF metamodels before doing this as mentioned before.
2.2.2 Architecture

The EAST-ADL2 plug-in can be seen as one component that is subdivided into three sub components (see Figure 6).

Figure 6 High Level View on the EAST-ADL2 RIF plug-in Architecture

1. Sub component RIFImport provides interfaces for invoking actions to fulfill the first two use cases (see 2.1.1 and 2.1.2) and represents the implementation for realizing these use cases. It represents the integration into the Eclipse framework (including dependencies to required Eclipse plug-ins), it realizes the user, it invokes the ATLLauncher to transform RIF into EAST-ADL2 and it serializes a new EAST-ADL2 model (first use case) or a refined EAST-ADL2 model (second use case) based on the result of the transformation via EMF.

2. Sub component RIFExport also provides interfaces and represents implementation, but for the third use case (see 2.1.3). It deserializes the current EAST-ADL2 model via EMF, prepares the loaded model for the requirements specification export (remember the RIF Export Area), calls the ATLLauncher to transform the new RIF Export Area into RIF based content and finally serializes via EMF the transformation result into a new RIF file.

3. Sub component ATLLauncher will be directly used through the RIFImport and the RIFExport. It is using services from the standard ATL Transformation Engine, but represents coarse grained service methods to the RIFImport and RIFExport (a service façade so to say). To do it so, the ATLLauncher already knows all needed metamodels for transformations (import/export) and all the options for running ATL transformations are also implemented static. Thus, the RIFImport resp. RIFExport does not need to know those informations and are thus completely decoupled from ATL specifics.

2.2.3 Detailed Realization

In this section, we will describe what exactly will be happen in the EAST-ADL2 RIF plug-in when processing each of the three use cases (white box view).
First, we will have a look at the \textit{RIFImport} sub component to describe inputs, outputs and activities to fulfill the first two use cases. Finally we will have a look at the \textit{RIFExport} sub component to describe the input, output and activities for the export use case.

For the first use case (see 2.1.1), the \textit{RIFImport} gets a RIF XML file containing the requirements data from which an initial EAST-ADL2 model should be derived and saved (see Figure 7). First, the \textit{RIFImport} sub component is deserializing the RIF XML file using the standard EMF services for XML based deserialization. The basis for doing that is the RIF EMF Ecore metamodel derived from the RIF XML schema. Thus, if the RIF XML file is valid to the RIF XML schema, it will also be valid to the RIF EMF Ecore metamodel.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure7.png}
\caption{Deriving a new EAST-ADL2 Model from a RIF Model}
\end{figure}

Now, the \textit{RIFImport} invokes services of the \textit{ATLLauncher} to transform the RIF model into an initial EAST-ADL2 model. For doing this, the \textit{ATLLauncher} needs the EMF based RIF model, the EMF based RIF metamodel, the EAST-ADL2 UML profile and the ATL transformation rules as inputs. The RIF metamodel and the EAST-ADL2 UML profile are the basis on which the ATL transformation rules have been developed.

E.g. the RIF root element in the RIF model will be transformed into two EAST-ADL2 elements. That is, the root package element which additionally is containing the Requirements Model. Elements directly and indirectly contained by the RIF root element will also be transformed as defined by the ATL transformation rules. This resulting EAST-ADL2 model elements will then be putted into the newly created Requirements Model in the same way.

Finally, after all ATL transformation rules have been processed, the resulting EAST-ADL2 model will be serialized into a XMI based UML file also using the appropriate EMF standard services.
The second use case (see 2.1.2) is slightly different to the first use case. The RIFImport sub component is even responsible for processing this. Figure 8 depicts the activities for his use case.

**Figure 8 Importing Requirements Data from a RIF File into an already existing EAST-ADL2 Model**

First, the RIF model file will be deserialized and the resulting model will be transformed as described for the first use case. But additionally, the RIFImport sub component got a reference to a file which contains the data of an already existing EAST-ADL2 model into which the RIF requirements data shall be imported. Thus, the RIFImport even deserializes the EAST-ADL2 model file. Now we have intermediately two models, the original EAST-ADL2 model and an EAST-ADL2 model resulting from the transformation of the RIF content. Both EAST-ADL2 model will be merged into a resulting EAST-ADL2 model (called Refined EAST-ADL2 Model). The merge will not be done by the ATL transformation engine. Instead it has been implemented with pure JAVA code. The merge mechanism adds a RIF import Area first and puts all data below the Requirements Model of the Intermediate EAST-ADL2 Model into this new area. Finally Multi-Level references will be established between elements of the Requirements Model and the new area. The user can see the final result after the RIFImport sub component has serialized the refined EAST_ADL2 model as outcome of the whole process.

The third use case (see 2.1.3) will be processed by the RIFExport sub component (see Figure 9). It gets as input the reference for the already existing EAST-ADL2 model file. It deserializes the data of this file and makes a query for getting all requirement top level structure elements (see RequirementContainer in the Domain Model). The top level structure of the complete requirements specification will be shown to the user GUI based. The user must chose exactly one of the provides top level structures. After he has choses one of the containers, the RIFExport sub
component continues with creating a new RIF Export Area inside the EAST-ADL2 model. Then, all Requirements data below the chosen container and the container itself will be copied (cloning) into this new area. This mechanism will be done by pure JAVA based implementation. Now, the RIFExport sub component takes the refined EAST-ADL2 model and invokes the ATL transformation for the direction EAST-ADL2 to RIF for the newly created RIF Export Area including all elements inside this area.

![Diagram](attachment://diagram.png)

**Figure 9 Export a Requirements Specification of an already exiting EAST-ADL2 Model into a RIF File**

E.g. the RIF Export Area will be transformed into a RIF model root element etc. The outcome of this transformation will be a new RIF model. The RIFExport sub component finally serializes the refined EAST-ADL2 model and the newly creates RIF model each into files.

### 2.3 Known Limitations
The current EAST-ADL2 RIF plug-in has been developed against RIF in version 1.1a. Thus, the current EAST-ADL2 RIF plug-in only supports RIF documents based on the RIF XML Schema for version 1.1a.

Due to missing concepts in the current EAST-ADL2 language, the EAST-ADL2 RIF plug-in does not support the metaclass RIF::Access. Thus, informations about whether a RIF element is read only or changeable etc. will be lost during the transformation from RIF into EAST-ADL2.

The EAST-ADL2 RIF plug-in currently does not support requirement links or groups on RIF and also not on EAST-ADL2 side.

The EAST-ADL2 language has several specializations for generic requirement element types. E.g. QualityRequirement as specialization of Requirement. To be flexible, the EAST-ADL2 RIF plug-in should be customizable, so that a user can specify RIF to EAST-ADL2 mapping properties. E.g. the user could express, that a RIF requirement with a user attribute ReqType and its value Quality will become a QualityRequirement on EAST-ADL2 side. Currently, the EAST-ADL RIF plug-in does not support such a user defined mapping. Instead, the mapping of such special typed elements is currently hard coded in the ATL transformation rules.
3 Tool Support for Dependability

This section describes the dependability analysis tool support for EAST-ADL2. Tool support is provided primarily by the HiP-HOPS tool with an EAST-ADL2 plugin to serve as the interface to HiP-HOPS. Both the plugin and the integration of HiP-HOPS with EAST-ADL2 concepts will be described below.

3.1 EAST-ADL2 Dependability Analysis Plugin

The EAST-ADL2 Dependability Analysis Plugin is a tool providing the means to perform dependability analysis on systems modelled using the EAST-ADL2 language. In EAST-ADL2, the user explicitly models the propagation of errors in a so called error model. The error model can be created easily because it is built from local error information. For system dependability, however, the global effects of the error propagations are of interest. Tools such as HiP-HOPS can calculate the global effects through dependability analyses like FMEAs and FTAs by combining the local error information. To provide a view of the global failure effects to EAST-ADL2 users without additional effort on the users’ side, we provide an automated link between EAST-ADL2 and HiP-HOPS.

3.1.1 Tool Integration

To establish the link between EAST-ADL2 design tools such as Papyrus and dependability analysis tools such as HiP-HOPS, they have to be integrated. Integration is described by Wassermann [X1] to have five different aspects:

- control integration: programs can interoperate;
- data integration: programs can use each others data;
- presentation integration: programs have a unified GUI;
- platform integration: services provided by platform;
- process integration: the software development processes are integrated.

The plugin mainly performs three forms of integration: data, control and presentation integration. All of these forms are realised in the form of a plugin for the Eclipse framework.
3.1.1.1 Control and Presentation Integration

This link is provided in the form of a plugin. This ensures seamless integration in the modelling environment (Papyrus/Eclipse) and keeps the dependability analysis overhead experienced by the user as low as possible and thus allows for an iterative development process.

Without tool support, dependability analysis requires tedious, manual work which is frequently seen as an obstacle by engineers, often reducing the scope of this task or limiting it to a single dependability analysis. Automation of dependability analysis has several advantages: it makes dependability analysis easier, it is readily available, and it allows the engineers to obtain a thorough and quick analysis of their design. This rapid feedback based on analysis results allows engineers to perform more micro iterations in the development process, where each iteration refines and improves the previously built model.

3.1.1.2 Data Integration

Data integration in this context is concerned with the transformation of error modelling data. We transform from an EAST-ADL2 representation to a HiP-HOPS representation, while preserving the semantics. State of the art data integration for model-based development is supported by powerful model transformation engines and languages. Different transformation languages and engines are available, each of them solving a particular problem especially well. Identifying the right model transformation language/engine for the task at hand is a fundamental part of the solution.

3.1.1.3 Transformation Design

We partitioned the model transformation process from EAST-ADL2 to HiP-HOPS into two steps. Each step has a separate purpose and concern.
Figure 11 - Model transformation process

(1) **Semantic Mapping Transformation**: The first transformation step transforms an EAST-ADL2 model that is created in the Papyrus UML modelling tool and creates an intermediate model. The structure of the intermediate model resembles the HiP-HOPS grammar, so it is close to the structure of the desired output. This stage performs the semantic mapping between the domains of EAST-ADL2 and that of HiP-HOPS. However, this stage is not concerned with the actual representation of the data.

(2) **Representation Transformation**: The second transformation step takes the intermediate model and creates the input file for the HiP-HOPS program. This step is mainly concerned with the representation of the information according to the concrete syntax required by HiP-HOPS.

In the following section we discuss the benefits of this solution.

- Our solution separates two different concerns of the transformation from EAST-ADL2 to HiP-HOPS: (1) the semantic mapping between the domains of EAST-ADL and that of HiP-HOPS and the (2) details of the concrete syntax of the HiP-HOPS input file.

- Each transformation is a separate, self-contained module, which can be developed, changed and tested independently. This decomposition into two separate transformations allows us to parallelise the work on the two transformations and reduce development time. It also allows the two transformations to evolve independently without affecting each other, e.g. a change in the HiP-HOPS grammar will only affect the representation transformation.

As discussed in the section on data integration, different transformation engines have different strengths which can be played out for different concerns. The solution allows us to select the best tool for each concern.

### 3.1.2 Semantic Mapping Transformation

The purpose of the Semantic Mapping Transformation is to map concepts from EAST-ADL2 to HiP-HOPS in a way that preserves the semantics of the original model. The mapping between EAST concepts and HiP-HOPS concepts is explained in the table below.
Figure 12 - Mapping of concepts from EAST-ADL2 to HiP-HOPS

Model to model transformations are well suited for a semantic mapping transformation. Both input and output of a model to model transformation are models themselves. Mapping patterns can be described by relational and declarative transformation languages in a concise manner. We chose the ATLAS Transformation Language (ATL) [X2], a language that allows a choice of relational and imperative constructs. It furthermore allows processing of models that have a profiled metamodel, i.e. a metamodel that consists of a metamodel and a profile description. In the case of EAST-ADL2 the metamodel consists of the UML metamodel and the EAST-ADL2 profile.

### 3.1.3 Representation Transformation

The purpose of the Representation Transformation is the generation of a textual description based on the intermediate model. The intermediate model is designed to have structure which is aligned to HiP-HOPS. No structural changes are required in this transformation. The focus is on serialising the model as text.

Textual representations can be generated particularly well with model to text transformation languages. We chose the Xpand language from OpenArchitectureWare [X3]. Xpand is a template-based model transformation language, which incorporates the output in the form of templates into the control structure. The intermediate model is explored using a depth-first strategy.

EAST-ADL2 models created in Papyrus have a metamodel that is a composition of several separate metamodels. In the case of EAST-ADL2, the metamodel consists of the UML metamodel and the EAST-ADL2 profile. These artefacts are composed by the Eclipse Framework to an EAST-ADL2 metamodel at runtime. The EAST-ADL2 metamodel corresponding to a Papyrus EAST-ADL2 model is not a separate artefact, and this complicates the model transformation and limits the choice of model transformation engines.
Figure 13 - Intermediate HiP-HOPS metamodel

The intermediate model conforms to the HiP-HOPS.ecore metamodel. It is aligned to the HiP-HOPS grammar. It also conforms to Ecore, and thus processable with the Eclipse Modelling Framework (EMF).
3.2 Extensions to the HiP-HOPS Dependability Analysis Engine

The EAST-ADL2 dependability analysis plugin creates an interface between EAST-ADL2 and HiP-HOPS that allows an EAST-ADL2 model to be analysed by HiP-HOPS. However, this interface relies upon a certain commonality of concepts between EAST-ADL2 and HiP-HOPS, which requires a degree of harmonisation. In addition, the extra modelling capabilities of EAST-ADL2 makes it possible to represent a wider range of dependability-related information than traditionally represented in HiP-HOPS, and thus to take fuller advantage of this information, HiP-HOPS also needs extending with new capabilities.

These processes are described below.

3.2.1 Harmonisation of EAST-ADL2 and HiP-HOPS

As EAST-ADL2 has evolved, it has incorporated information necessary to support a dependability driven model-based design process. In particular, it has been designed to be able to support the upcoming ISO°26262 automotive standard, which sets out a methodology for the incorporation of dependability in the design of an automotive system. But EAST-ADL2 is only a modelling language – to use the dependability information contained within an EAST-ADL2 model, an external tool is needed. HiP-HOPS is one such tool – a recently developed dependability analysis & optimisation engine [4] designed to support model-based dependability analysis techniques. However, HiP-HOPS was not originally designed to be compatible with EAST-ADL2 or ISO°26262, and assumes a somewhat different modelling approach. To ensure that EAST-ADL2 and HiP-HOPS can work together, the dependability-related concepts in both need to be harmonised.

As explained earlier, EAST-ADL2's primary means of representing dependability information in a model is via its Error Model. The Error Model is designed to be a separate architecture, parallel to – but not necessarily with a one-to-one mapping to – the nominal architecture. By contrast, HiP-HOPS uses a hierarchical model that represents both nominal architecture and failure behaviour together. This is the first major difference with HiP-HOPS. EAST-ADL2's Error Model allows a great deal of freedom when modelling the failure behaviour of a system, whereas HiP-HOPS uses a much more constrained model; although the unconstrained Error Model can in theory be converted to an equivalent HIP-HOPS model, HiP-HOPS expects that model to represent the structure of the nominal system architecture, not just a 'virtual' error architecture. Though this does not necessarily preclude a successful analysis, it does require a certain degree of careful interpretation of the results, since the components that HiP-HOPS refers to are really entities in the EAST-ADL2 error model, and not nominal entities. This issue takes on much greater importance when optimisation is being applied, in which case the EAST-ADL2 Error Model must be constrained to a one-to-one mapping with the corresponding nominal model for the optimisation to make sense. This has required the development of methodological guidelines and the provision of superimposed stereotypes to allow nominal and error models to share the same architecture for the purposes of generating models that can be optimised by HiP-HOPS.

The modelling concepts of HiP-HOPS have also informed the design and development of the EAST-ADL2 Error Model itself. HiP-HOPS was designed with a compositional model that allows hierarchies of subsystems and subcomponents to be built up, all with ports (interfaces to the rest of the system) and interconnecting lines (connections between ports). Each component is defined with data describing its local failure behaviour; in particular, 'failure logic' is defined that relates failures at a component's outputs ('output deviations') to a combination of internal failure modes ('basic events') and failures received at the component's inputs ('input deviations').

As the plugin's concept mapping table shown earlier indicates, many of these concepts (or their equivalents) have found their way into EAST-ADL2. The failure logic is contained in EAST-ADL2's ErrorBehaviour entities, while ErrorModelTypes and ErrorModelPrototypes build up a hierarchy in the same way HiP-HOPS uses subsystems and components. FaultFailurePorts behave similarly to
HiP-HOPS ports and their associated output/input deviations, and are connected by error propagation links analogous to HiP-HOPS lines. InternalFaults represent internal failure modes.

Thus the development of the Error Model in EAST-ADL2 has evolved with an improved level of support for HiP-HOPS concepts in mind, particularly with regard to system failure propagation. However, HiP-HOPS is only one possible tool and so care has been taken to ensure that EAST-ADL2 dependability concepts are not limited to HiP-HOPS; indeed, in theory the Error Model should also be transformable to other compositional dependability analysis techniques or even more formal techniques. In some cases this has led to complications in the translation to HiP-HOPS (e.g. in the requirement for a one-to-one mapping in optimisation), but the benefit is a greater degree of flexibility and potentially a wider range of supported tools.

### 3.2.2 Extensions to HiP-HOPS

Although the dependability analysis concepts from HiP-HOPS have influenced the development of the Error Model, resulting in a closer harmonisation between the two, there are some concepts found in EAST-ADL2 – particularly in terms of new support for ISO26262 processes – that have no equivalent in HiP-HOPS. Thus to support these concepts, HiP-HOPS has also needed extending with new capabilities. The two major extensions are to support multi-perspective analysis and the decomposition of dependability requirements.

#### 3.2.2.1 Multi-Perspective Analysis

As explained above, EAST-ADL2 does not limit the modeller to a single architecture, as HiP-HOPS does. Not only are the Error Model and nominal model separate, but EAST-ADL2 provides multiple layers or levels of potential modelling as well as different views or perspectives of a model. HiP-HOPS is expected to be applied to EAST-ADL2 models at the analysis/artefact and design levels, and at these levels EAST-ADL2 offers a number of different perspectives. At the analysis level, the primary perspective is the Functional Analysis Architecture (FAA), which is a relatively abstract view of the functions of the system. At the design level, the model is more concrete and is separated into the Functional Design Architecture (FDA) or software perspective and the Hardware Design Architecture (HDA) or hardware perspective. There is also the possibility of a middleware perspective at the design level.

Although HiP-HOPS is designed to be able to support iterative analyses at different levels of abstraction, and thus can cope equally well with an initial qualitative analysis of an abstract functional model and a later quantitative analysis of a more detailed component model, it was only designed to support a single perspective. The concept of separating software and hardware perspectives of the same system is without analogue in HiP-HOPS. This is because HiP-HOPS assumes a compositional model that combines hardware and software, with individual software functions (or subsystems of functions) contained within the hardware processors that run them. Therefore, although HiP-HOPS is capable of analysing both software and hardware, it requires them to be in the same model, with the hardware the primary means of failure propagation: it is not possible for failures in one software function to propagate directly to another remote, physically separated software function; instead software failures must propagate up to their parent hardware component and then propagate along hardware connections to other hardware components that carry the affected functions.

This can be emulated in EAST-ADL2 by combining everything into a single perspective, but the advantages of having multiple perspectives are then lost. Instead, HiP-HOPS is being extended to provide native support for multi-perspective analysis: the analysis of failure behaviour across two or more separate perspectives of the same system (e.g. H/W and S/W). This requires the introduction of the concept of a 'perspective' to the HiP-HOPS model hierarchy.
In EAST-ADL2, hardware and software entities are in separate architectures (HDA and FDA) and communication from one perspective to the other is accomplished by means of allocation relationships: software functions are allocated to hardware components that execute them. Failures of the hardware components will propagate to the software functions allocated to them. Failures can also propagate from H/W to H/W and also from S/W to S/W, but this form of connection is restricted to only H/W $\rightarrow$ S/W propagations. This avoids problems inherent in the multi-perspective modelling paradigm where failures can initiate intractable circular logic loops in when propagating through the system (e.g. a S/W fault could cause a H/W failure that in turn is caused by the same S/W fault occurring).

HiP-HOPS has been extended with a similar approach, as shown in Figure 14. Although the original single-perspective approach is still possible, when analysing FDA/HDA models from EAST-ADL2, the new multi-perspective representation can be used. To accomplish this, a new 'Perspective' layer in the model hierarchy was added:

- **Model** (top-level entity representing an entire system under analysis)
- **Perspective** (contains a different perspective of the entire system, e.g. S/W, H/W)
- **Subsystem** (subsystem containing one or more components)
- **Component** (component representing some H/W or S/W entity)
- **Implementation** (the failure behaviour of a component; may also contain a subsystem)

Any number of perspectives can be used, but one perspective must be the 'primary' perspective. Since EAST-ADL2 assumes that the FDA is the primary perspective, and restricts failures to propagating from H/W to S/W but not from S/W to H/W, it is assumed that an exported EAST-ADL2 model will use the software perspective as the primary perspective in HiP-HOPS too.

As in EAST-ADL2, HiP-HOPS also provides means for failures to propagate from one perspective to another. As in EAST-ADL2, it is possible to define 'allocations' of components in one perspective to components in another perspective. Failures can propagate along these allocations by means of
'External Propagation' constructs. In fact, in both EAST-ADL2 and in HiP-HOPS, it is possible to define more than one possible allocation, e.g. using variability constructs in EAST-ADL2. This allows scope for potential optimisation of the model on the basis of changing the allocation of software functions to different hardware components, and indeed the optimisation capabilities of HiP-HOPS are also being extended in that direction.

HiP-HOPS provides for certain types of 'local failure data'. Traditionally this has included definitions for Basic Events (representing internal failure modes), Output Deviations (representing the propagation of failures at component outputs), and Common Cause Failures (linked to common model-level failures). Each, once declared, is visible to other definitions in that component, e.g. a basic event can be used in an output deviation of the same component.

Due to multi-perspective analysis, HiP-HOPS now also supports the definition of 'External Propagation' failure data. These are equivalent to Output Deviations, but rather than representing the propagation of failure from the component outputs, they represent the propagation of failure across allocation relationships. Also, unlike other failure data types, External Propagations are visible to allocated components as well as the local component. This means that they can be used in the definition Output Deviations of allocated components.

For example, consider CPU2 and F2 in Figure 14 above. We may define the local failure data of CPU2 to include:

- Basic event "CPU Failure", optionally with appropriate failure rates etc.
- Output Deviation "Omission-output", which is caused by either "Omission-input" or "CPU Failure" (defined above) or "EMI" (defined below).
- Potential Common Cause "EMI", which is caused by electromagnetic interference affecting the entire system.

In addition, because we know that CPU2 can have software functions allocated to it, we define an External Propagation like so:

- External Propagation "Omission", which is caused by "Omission-input", "CPU Failure", or "EMI".

Then when we define the local failure data for F2, which is a software function allocated to CPU2, we can reference this external propagation in F2's output deviation:

- Basic event "Undetected bug"
- Output Deviation "Omission-fout", caused by "Omission-fin" or "Omission". The external propagation is referred to more explicitly as "FromAllocation(Omission)".

Because any component can only be allocated to one component at a time (it has at most one 'current allocation'), we know that "Omission" must refer to an external propagation found in CPU2. HiP-HOPS can therefore link this "Omission" in F2 to the failure logic for "Omission" in CPU2 during its fault tree synthesis process, connecting the perspectives together and allowing for propagation of failure from hardware to software.

Note also that the semantics of common cause failures have been changed too; whereas before common cause failures (CCFs) were globally visible throughout the model, now CCFs are defined...
per perspective, thus a CCF defined for the hardware perspective (e.g. flooding, fire) would not be accessible from a component in the software perspective, for instance.

HiP-HOPS also allows a more flexible form of connection by means of the 'LocalGoto' and 'GlobalGoto' declarations. These support a more abstract type of connection for situations where there is no form of allocation relationship between two components in different perspectives, but propagation is nevertheless necessary. In these cases, the Gotos act like special forms of input deviations that can connect directly to another output deviation elsewhere in the model. Local gotos only connect to output deviations in the same subsystem (hence 'local') whereas Global gotos can connect to any fully-qualified output deviation in any perspective. For example:

- "O-F2.out = LocalGoto(O-F1A.out)" is valid and states that O-F2.out is caused by O-F1A.out.
- "O-F2.out = LocalGoto(O-CPU1.out)" is invalid as the output deviations are not in the same subsystem.
- "O-F2.out = GlobalGoto(O-CPU1.out)" is valid and states that O-F2.out is caused by O-CPU1.out.

However, these gotos are considered potentially harmful to a coherent analysis and should only be used sparingly, because they break the connection between the propagation of failure and the architecture of the system.

Because these new connections (allocations and gotos) connect existing HiP-HOPS constructs (namely, output deviations), they fit relatively seamlessly into the HiP-HOPS fault tree synthesis process. Once the fault trees are generated, then they can be analysed as normal without any further distinction between one perspective and another.

It is hoped that, once complete, these extensions to HiP-HOPS will enable better support for the analysis of EAST-ADL2-style multi-perspective models involving separate representation of hardware and software.

3.2.2.2 Decomposition of Dependability Requirements

One area where EAST-ADL2 is being developed to offer support for an ISO 26262 process is in the area of dependability requirements. A hazard analysis on vehicle feature level is able to determine the possible hazards for a system and assign corresponding safety goals, each of which includes the requirement to reach a certain level of safety known as an Automotive Safety Integrity Level or ASIL. In ISO 26262, these safety requirement (i.e. ASILs) can be decomposed through the system to obtain safety requirements for those parts of the system that cause any given hazard, rather than making the entire system meet the requirement (which may not even be possible).

EAST-ADL2 provides the means to support the hazard analysis on vehicle level and also allows the results (in the form of ASIL requirements) to be passed down to the analysis and design levels, where they can be associated with causes in the Error Model, thus linking the propagation of individual failure modes through the model with the hazards they cause and thus the ASIL required to prevent that hazard.

However, once again, although EAST-ADL2 can model the relevant information, it cannot actually perform decomposition of ASILs itself. Nor can HiP-HOPS, which is was not designed with that sort of analysis. In fact there are no tools which can perform this activity, and perhaps no other language is designed to represent it. ASIL decomposition is a new concept and requires new modelling entities and tool algorithms to support it.

As a result, HiP-HOPS is being extended with new algorithms to support ASIL decomposition. The goal is to be able to decompose the safety requirement (represented by the ASIL) associated with
a particular hazard and determine which safety requirements are necessary for individual failure modes to fulfil the safety requirements of the system as a whole. For example:

In this example, the modelled system as a whole has ASIL C, indicating there is a relatively severe hazard that must be avoided. Parts of the system that contribute directly to the system failure (such as the Dedicated CAN and the outputs) receive the ASIL directly too. When there is a disjunction, e.g. two or more components can all cause the system failure individually, they all receive the full ASIL. Only when components must fail together in conjunction to cause the system failure is the ASIL diluted. This can be seen in the simple case for Macro-block3 in the diagram, which collectively receives an ASIL C; the subcomponents within (ultimately SWS1-4) may not have the full ASIL C, however, and there are a number of different possible combinations of ASILs that would fulfil the ASIL C for the subsystem as a whole (e.g. A + A + A + A).

A more complex case arises with Macro-block1 and Macro-block2. Both must fail in conjunction to cause system failure, thus both together receive a share of the ASIL C. In the figure, Macro-block2 is assigned ASIL B whereas Macro-block1 is assigned ASIL A. However, both blocks have two outputs; in Macro-block1, a failure of either output is all that is necessary, whereas in Macro-block2, a failure of both outputs is needed. In Macro-block1, all components would require ASIL A since any individual failure is sufficient, but in Macro-block2, both originating contributors (NO and NC in the EPB Button subsystem) could receive ASIL A as both need to fail for Macro-block2 to fail.

In this example, there are many alternative ASIL allocation strategies that could be used. However, when there is more than one hazard and thus more than one system safety requirement, the competing demands of these multiple requirements can be used to reduce the number of possible allocations. In the more abstract example below, there is only one optimum allocation possible:
Here, there are four possible hazards linked to four possible system failures, each of which has been given a different safety goal (and associated ASIL):

- F1(O) - Omission of output from F1, with ASIL D
- F1(C) - Commission of output from F1, with ASIL A
- F2(O) - Omission of output from F2, with ASIL C
- F2(C) - Commission of output from F2, with ASIL A

Due to the connecting failure logic amongst the six subcomponents in the system, it transpires that there is only one possible allocation. To see why, it is necessary to go into more detail. First, the causes of each hazard must be explored. As an example, take F1(O). Its initial cause is an omission from E1, which can be caused by an internal failure mode or by an omission of input. Since E1 is directly capable of causing the hazard, it receives ASIL D. Omission of input to E1 is caused solely by an omission from E2, which in turn is caused either by an internal failure mode or by an omission of input at both inputs. E2 therefore receives ASIL D as well. However, omission of input to E2 is caused by a conjunction of omissions from both E3 and E4, so they each contribute some share of the ASIL D. There are five possible allocations for these two components:

- E3 = QM (0), E4 = D (4)  Total = 4 (D)
- E3 = QM (1), E4 = D (3)  Total = 4 (D)
- E3 = QM (2), E4 = D (2)  Total = 4 (D)
- E3 = QM (3), E4 = D (1)  Total = 4 (D)
- E3 = QM (4), E4 = D (0)  Total = 4 (D)

Allocations where both E3 and E4 are more than sufficient to meet the safety requirement, e.g. E3 = ASIL C and E4 = ASIL C, are also possible, but not considered unless necessary (i.e. unless both E3 and E4 directly contribute to an ASIL C hazard somewhere).
To reduce the number of combinations, it is necessary to look at the other ASILs. It is determined through decomposition that F1(C), with ASIL A, is caused by any internal failure mode of E1, E2, E3, and E4; thus each of those failure modes must be at least ASIL A. That eliminates two possibilities for E3 and E4 (namely, where one is ASIL D and the other is QM). Next, decomposition of F2(O) – with ASIL C – shows that there are four possible causes: internal failures of E2, E4, E5, or E6. Each therefore requires a minimum of ASIL C. This removes two more possibilities from the list, leaving only one possible allocation: E3 = ASIL A, E4 = ASIL C.

In this way, ASIL decomposition allows us to determine which failure modes contribute to which hazards, and by process of elimination, determine what ASIL each failure mode should have (or, more frequently, determine a set of many possible ASIL allocations for those failure modes). The new algorithms in HiP-HOPS support this.

The first step is to ensure that the failure behaviour of the system is also modelled in an Error Model. This allows HiP-HOPS to understand the routes of propagation through the system necessary to cause each hazard, each represented by a different fault tree. Fault Tree Analysis then yields the combinations of basic failure modes necessary to cause each hazard, and these are assigned ASILs accordingly. For example, if we have a fault tree with the following cut sets (combinations of failure mode causes):

- \{A \cdot B \cdot C\}
- \{D\}
- \{E \cdot F\}

and the fault tree leads to a hazard with associated ASIL C, then each set receives ASIL C as well. In cases where more than one failure mode is involved (e.g. A.B.C and E.F) then multiple possible allocations to those failure modes are possible. For example, all of A, B, and C may receive ASIL A, or one of them may receive ASIL C and the others only QM. It is up to the designer to decide which possible allocation is best.

However, as previously explained, HiP-HOPS can use multiple system safety requirements to reduce the number of possible combinations by eliminating impossible ones (i.e. allocation combinations in which one or more safety requirements are unfulfilled) and ignoring ones that are unnecessarily excessive (i.e. ones that assign higher ASILs to failure modes than are required). This helps the designer to make an informed choice, although there may still be many possible allocations to choose from.

This ASIL decomposition process is designed to be applied to the FAA on the analysis level. At this stage the model is still quite abstract, and true internal failure modes may not be known, but it can be assumed that each function has at least one internal failure mode that causes any given output failure. Thus for example function X may have an omission failure mode, a commission failure mode, a value failure mode, and so forth. Since these failure modes will often directly be responsible for causing output failures of their functions, assigning an ASIL to the failure mode is synonymous with assigning an ASIL to the output failure of the function too. This is not what matters, however: what is important is that the ASIL is assigned on the basis of that component's local contribution to the system failure, and only includes a consideration of input to that component if it mitigates the failure in some way (e.g. it requires all inputs to fail in conjunction, or fails silent in response to error).

The result should be an allocation of potential ASILs to the failure modes (or more generally, the functions) of the FAA. Then on the next level of development, when the design is more mature and an FDA has been modelled, these preliminary ASILs can be tested by conducting a full quantitative analysis on the FDA (and optionally the HDA etc too in a multi-perspective analysis). This will yield probabilistic values for failures that can be used to determine whether the originally assigned ASILs have been met; for example, guidelines may state that an unavailability of 1x10^-5 or less is
sufficient to meet ASIL B, whereas $1 \times 10^{-6}$ is needed to meet ASIL C. If safety requirements are not being met, then the system design can be changed accordingly until they are met. Alternatively the original allocation of ASILs may be questioned, and another allocation strategy chosen instead.

This ASIL decomposition capability in both EAST-ADL2 and HiP-HOPS is a particularly novel and potentially very valuable capability that is thus far unique. It could prove to be a very effective means to support a safety-driven design process.
4 Tool Support for Variability

This chapter provides an overview of the variability-related plugins used or developed in ATESSST2. The following chapter then describes, from a wider angle, how they fit together and how this forms an ensemble that can be used for variability and product line management.

The tool support for variability in EAST-ADL2 was extensively refined in Q1 and Q2 of 2009 according to the review of EAST-ADL2 variability concepts in WT3.3 in Q4 of 2008. From Q3 of 2009 on, the main focus was on consolidation, improvement of usability and addition of new ideas that resulted from experiments and evaluation efforts in ATESSST2. The extensive study of the HAVEit demonstrator is particularly noteworthy in this respect and led to numerous improvements and corrections of the tools. At time of writing the current release of this tool support is 0.6.1.

Compared to the status at the end of ATESSST1, the tool was renamed from “IoVM” to “CVM Framework”, which stands for “Compositional Variability Management Framework”, highlighting the novel approach to handling variability in composition hierarchies of FunctionTypes, as employed in ATESSST2 and EAST-ADL2.

This tool support for variability is provided in two major parts:

1. **CVM Variability Management Framework**: core CVM with Model Editor, textual VSL Editor, etc.
2. **EAST-ADL Bridge for CVM Organizer**: EAST-ADL related functionality
3. **EPM Function Editor**

No. 1 is a stand-alone version of CVM covering the fundamental variability management concepts of EAST-ADL2 but without relying on the rest of EAST-ADL2. No. 2 then integrates CVM into the EAST-ADL2 language and framework, using the EAST-ADL2 profile from CEA as a basis. This also allows to use CVM together with Papyrus. No. 3 is an experimental domain specific editor for editing the core of EAST-ADL2 elements, esp. FunctionTypes and their internal structure. It was implemented to experiment with various editing related aspects of variability management in EAST-ADL2, for example: how can the editing of a FunctionType’s public feature model and internal binding be best supported with editing and visualization functionality in a DSL editor.

The plugin is available on the ATESSST2 SVN in folder /WP3/3.3/CVM/

A short movie demonstrating the plugin and its use together with Papyrus is available on the ATESSST2 SVN in folder /WP3/3.3/Misc (also some screenshots can be found there).

Further information on the plugin is available on the web-site [http://www.cvm-framework.org/](http://www.cvm-framework.org/)

4.1 CVM Framework Usage

This chapter provides a quick overview of the core functionality of CVM without going into great detail. It also shows how the different parts of CVM - the model editor, diagram editor and the VSL editor - fit together.

**Prerequisites**

For this tutorial, you need an installation of Eclipse with CVM, either as a stand-alone RCP version or as an Eclipse plugin. For more information on installation please refer to the CVM manual, section "Installation".

1. **Creating a New File**
We start by creating a new model file. Such model files have the extension ".cvm".

<table>
<thead>
<tr>
<th>Plugin Version</th>
<th>RCP Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. In Eclipse, select from the main menu &quot;File / New / Other ...&quot;.</td>
<td>1. In the RCP version, select from the main menu &quot;File / New CVM File ...&quot;.</td>
</tr>
<tr>
<td>2. In the dialog that opens up, select file type &quot;CVM Variability Model&quot; from category &quot;CVM&quot;; then click &quot;Next&quot;.</td>
<td>2. Choose a folder and file name in the dialog and click &quot;Save&quot;.</td>
</tr>
<tr>
<td>3. Choose a container (i.e. an Eclipse project to contain our new file).</td>
<td>3. Select to create an empty file.</td>
</tr>
<tr>
<td>4. Change the file name to &quot;tutorial1&quot;.</td>
<td></td>
</tr>
<tr>
<td>5. Select to create an empty file.</td>
<td></td>
</tr>
<tr>
<td>6. Click &quot;Finish&quot;.</td>
<td></td>
</tr>
</tbody>
</table>

This opens up the CVM model editor showing our newly created file. As you can see, the file has already been populated with a single root element of type VariabilityModel. Such a variability model is the root container for all information managed by CVM.

The default name for this is "New Variability Model". To change this name, double-click on the name of the element in the model editor and type "VM" and press Enter.

2. Creating a Simple Feature Model

To add content to our variability model we will now create some Features. Features cannot be contained directly in a variability model; instead we first have to create a FeatureModel.

1. Right-click the variability model element, now called "VM", and select "New Child / Feature Model" from the context menu.
2. Expand the variability model element (by clicking on the "+" symbol).
3. As above, rename the newly created feature model by double-clicking on its name. Call it "FM1".

Now, your editor window should look similar to this:

```
<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VM</td>
<td></td>
</tr>
<tr>
<td>FM1</td>
<td></td>
</tr>
</tbody>
</table>
```

In this feature model, we now define the variability of a wiper in a fictitious car product line:

**Example 1:** The wiper comes in two core variants: a simple and an advanced form. In the second case, more sophisticated interval modes are provided. In addition, the wiper may provide an automatic mode in which wiping is controlled depending on the current rain intensity. All our cars have such a wiper.

To create the features for this example, you have several options:

- Right-click the parent feature (or feature model) and select "New Child / Feature" from the context menu.
- Select a feature (or feature model) and press ALT+C to create a child feature or press ALT+S to create a new sibling feature.
- Select a feature (or feature model) and click the button on the CVM tool bar.
Obviously the simple and advanced forms of our wiper are alternative, so we need a feature group of cardinality [1]. To create such a feature group you again have several options:

- As for features above you can use the "New Child" context menu action, use a keyboard short-cut (ALT+G for create group, in this case), or the tool bar.

- Alternatively, you can select two or more sibling features which are not grouped yet and select "Group Features" from the context menu. This will create a new feature group and add the selected features in a single step.

Finally, to set cardinalities use the context menu and to edit the feature’s names and descriptions double-click on them in the model editor.

The complete feature model for the above example then looks like this:

![Feature Model Diagram](image)

Note that we have set feature "Wiper" and the feature group to cardinality [1].

3. Feature Modeling in the Diagram Editor

To illustrate configuration links later in this tutorial, we want to add a second feature model. We could do this as before for "FM1", but let's use the diagram editor for this.

This is what we want to model:

Example 2: In contrast to the rather end-customer oriented variability specification in feature model "FM1" above, we now want to specify our wiper's variability on a more technical level. It may have different forms of wiping modes and may have a rain sensor. A continuous wiping mode is always present. In addition, an arbitrary number of interval modes can be chosen during configuration. Finally there is the option of a flexible interval mode, where the driver can adapt the duration of the wiping interval by rotating a special button.

First, we need to create a variability diagram. So:

1. Right-click the root variability model and select "New Child / Variability Diagram" from the context menu.

2. Rename the new diagram to "VD".

3. To now open the diagram editor, please right-click our diagram "VD" and select "Open Diagram" from the context menu. This opens a new editor window showing an empty diagram.

4. Make sure to expand the tool palette on the right side of this window; you might have to click on the little arrow on the right side.

If everything went smooth you will see something like this:
Creating a feature model here in this diagram is pretty straightforward:

1. In the palette, click on the tool for creating feature model (called "Feature Model", located in the "Elements" tool box).
3. The name is already selected for direct editing. Type "FM2" and press Enter here.
4. Enlarge the feature model's visual to make room for contained features, by clicking and dragging the lower-right corner of it. (NOTE: In the future, you can create and resize an element in one go by clicking and dragging in step 2 above.)

In the same way create 6 features and rename and arrange them as shown in the following screenshot (don’t resize them, they will resize automatically if no size is set by the user).

Now add parent/child relations as follows:

1. Click on the "Parent / Child" tool in the palette.
2. Click on the parent feature (do not drag!).
In this notation, optional features are marked with a white circle and mandatory features are marked with a black circle (the usual FODA notation by Kang et al.). As you can see, all features are optional. Use the context menu by right-clicking the features in order to ...

- set the cardinality of feature "Continuous" to [1], i.e. mandatory and ...
- set the cardinality of feature "Interval" to [0..*], i.e. cloned. Note how this cardinality, which is not available in FODA feature modeling, is presented in the diagram.

Your editor window should now show something like this:

![Diagram of feature model](image.png)

The diagram as presented in this screenshot could be saved to an image file. To do this, make sure nothing is currently selected in the diagram and then right-click the diagram's background. Select "Save to Image ..." from the context menu and choose a file name and image format in the dialog.

We won't use the diagram editor for the remainder of this tutorial, so you can close it now.

4. Parameterized Features and Feature Links

Before proceeding, we first refine our feature model "FM2" a bit further, to make the discussions later in this tutorial, esp. on configuration links, more interesting.

The feature "Interval" with cardinality [0..*] represents an interval mode with a certain, fixed duration. However, we cannot define the actual duration of such an interval yet. To achieve this, make "Interval" a parameterized feature of type Float:

1. Select feature "Interval".
2. If the properties view of Eclipse is NOT visible already: Right-click feature "Interval" (in the model editor!) and select "Show properties view" from the context menu.
3. In the properties view choose tab "Basic" and set a check-mark on option "Parameterized Feature" (you might have to scroll down to see this option).
4. Select "Float" as the type (still in the properties view). The parameter section in the properties view for feature "Interval" should now look like this:
As you can see, we could also define a minimum and maximum value here or provide a default value. But we do not want this in our case.

5. Add an informative description to feature "Interval" by double-clicking the corresponding cell in the model editor.

At this point you should have this:

Let us now add a final detail to our feature model: a dependency. For this purpose, we assume that feature "RainSensor" may not be combined with the "Flexible" interval mode (for some technical reason). Such a dependency or constraint is realized in CVM by way of a FeatureLink:

1. Select feature "RainSensor".

2. If the properties view of Eclipse is NOT visible right now: Right-click feature "RainSensor" (in the model editor!) and select "Show properties view" from the context menu.

3. In the properties view choose tab "Links" --> the properties view now shows a list of feature links to/from feature "RainSensor" (should be empty for now).

4. Click on button "Create ..." (in the properties view).

5. In the dialog that shows up: select the target feature for the new feature link, i.e. "Flexible" in our case.

6. Click "Ok" --> the new feature link should now have appeared in the list.

7. Change the type of the newly created link to "excludes" and set a check-mark in column "<->" to make it bidirectional (both can be edited directly in the table).

Your properties view should list a single feature link now, as shown here:
5. Configuring Feature Models

So let us configure this feature model now:

1. Right-click feature model "FM2" in the model editor and select "Configure ..." from the context menu.
2. Edit the configuration in the configuration dialog that shows up (see screenshot below).
3. If you click "Ok" on leaving the dialog, then a new feature configuration will be created that captures the configuration edited in the dialog.

If you right-click an existing feature configuration instead of a feature model and select "Configure ..." from the context menu, then you can edit this existing configuration instead of creating a new one.

6. Configuration Links

Several feature models can be linked with respect to their configuration by way of a so-called configuration decision model. This means with the information captured in a configuration decision model it is possible to derive configurations of one feature model, called target feature model, from valid configurations of another, called source feature model. Such an arrangement of source and target feature models linked together by a configuration decision model is also called a configuration link.

The actual elements for defining how to configure the target feature model depending on the configuration of the source feature model are called configuration decisions. As the name suggests, a configuration decision model is made-up of many configuration decisions.
To illustrate this, let us create a simple configuration decision model:

1. In the model editor, right-click the root variability model "VM" and select "New Child / Config Decision Model" from the context menu.
2. Double click the newly created element's name and enter "CDM" as the new name.
3. We now have to define the source and target feature models linked by our new config decision model: Right-click element "CDM" and select "Edit Source & Target Feature Models" from the context menu.
4. In the dialog that opens, select "FM1" from the tree view; then click button "Add as Source"; in the "Set Name" dialog that opens up, confirm by clicking button "Ok" without entering a name.
5. Similarly, select "FM2" from the tree view and click button "Add as Target" and again confirm with "Ok".

Now you should see something like:

Note that we could provide local names for the source and target feature models that reflect their precise role within the configuration link we are currently defining. In our simple example, however, we do not need this. (Remark: with such explicit naming we can even use the same feature model more than once as source or target, for example to create two distinct target configurations for the same feature model.)

6. Click button "Ok" in the dialog to get back to the main model editor.

We have now created a configuration decision model called "CDM" with "FM1" as source and "FM2" as target feature model. The next step is to actually define how to configure "FM2" depending on a given configuration of "FM1":

1. Right-click "CDM" and select "New Child / Config Decision" from the context menu. This creates a new configuration decision, i.e. a conditional rule how to configure "FM2". For the next steps, please make sure you are in the model editor's grid view by clicking the 'Grid' tab on the lower side of the editor window.
2. In the row of the new element, double-click the cell in the first table column and enter "Advanced OR Automatic". This is an expression in propositional logic on the configuration of "FM1" that states when this rule shall trigger, i.e. have an effect on the configuration of "FM2".
3. In the row of the new element, double-click the cell in the table column called "Included Features" and enter "Flexible". This is the configuration decision's effect in the form of a list of features to be selected whenever the condition in column one (cf. step 2 above) evaluates to true. In this case we just state that feature "Flexible" is to be selected.

4. Repeat steps 1 through 3 to obtain five configuration decisions with the following information:

<table>
<thead>
<tr>
<th>Criterion (step 2)</th>
<th>Effect (step 3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>#1 true</td>
<td>Interval$short, short:Interval=1</td>
</tr>
<tr>
<td>#2 true</td>
<td>Interval$long, long:Interval=4</td>
</tr>
<tr>
<td>#3 Advanced</td>
<td>Interval$medium, medium:Interval=2.5</td>
</tr>
<tr>
<td>#4 Advanced OR Automatic</td>
<td>Flexible</td>
</tr>
<tr>
<td>#5 Automatic</td>
<td>RainSensor</td>
</tr>
</tbody>
</table>

5. The criterion "true" is used to define defaults that are always applicable. The effects of configuration decisions #1 to #3 are special in that they not merely select an optional feature but create instances of cloned features and assign values to them.

Now, your editor window should look something like this:

With these configuration decision we have defined how to configure "FM2" depending on a given configuration of "FM1": we always have two default intervals "short" and "long"; the advanced version of the wiper gets an additional medium interval; the automatic variant, represented by feature "Automatic" in "FM1" gets a rain sensor; finally, the flexible interval denoted by feature "Flexible" in "FM2" will be selected both for the advanced version of the wiper and the automatic variant (and the case of an automatic advanced wiper, of course).

To try out interactively this newly defined configuration link you use a dedicated view of CVM called "Configuration Preview":

1. If the configuration preview is not open yet, select "Window / Show View / Other ..." from the main Eclipse menu. In the dialog select "Configuration Preview" and click "Ok".

2. Place this view below the main editor window.

3. Select configuration decision model "CDM" in the editor. (Note: do not select an individual configuration decision because this will put the configuration preview in editing mode.)

The preview now shows the source feature model "FM1" on the left and the target feature model "FM2" on the right:
You can edit the left side and the right side will automatically be updated with the configuration derived from the one you edited on the left. The above screen-shot shows the configuration of "FM2" for the Simple variant of the wiper with "Automatic" feature: no special medium interval but "Flexible" and "RainSensor" are selected. A yellow highlighting the effects of the configuration link took place. Play around with the configuration on the left and verify that the configuration on the right is updated correctly through the configuration link.

CVM provides much more support for editing configuration decisions and for automatically deriving target configurations from given source configurations. Furthermore, it is possible to define configuration graphs, i.e. complex networks of configuration in which the nodes represent feature models and the edges are realized by configuration decision models. However, this detail is beyond the scope of this overview tutorial.

7. Textual Variability Specification

Finally, let us examine how our sample variability model would look like in the textual variability specification language VSL.

For this, we can now close the model editor. So,

**Plugin Version**

1. Close the CVM model editor.
2. In the view called "Navigator" or "Package Explorer" find the ".cvm"-file we have created above.
3. Right-click it and select "Convert to .vsl" from the context menu, --> a new file with the same name but extension ".vsl" will be created.
4. Now double-click this new "vsl" file, --> the VSL editor will open.

This is what you should see now:

**RCP Version**

1. **NOTE:** In the RCP version, the context menu action for converting from ".cvm" to ".vsl" files is not available yet. Therefore you need to create a new ".vsl" file with a sample model for now, so
   Select from the main menu "File / New VSL File ...".
2. Choose a folder and file name in the dialog and click "Save".
3. Select to create a file with a sample model.
NOTE: The code in this screenshot was simplified for illustration purposes.

In the editor window that opened up, you can see the VSL code of our small example from above. We won't go into detail of the syntax of VSL here, but you can experiment with the syntax in the editor. Whenever you save the file, the syntax will be checked and errors will be highlighted.

8. Summary

We have seen how to create feature models in the model and diagram editor and how to configure them and store the configurations. We have also examined how to link two feature models with respect to their configuration by way of configuration links. Finally, the alternative textual representation using the variability specification language was briefly presented.
5 Tool Support for Behavior modelling

5.1 Engineering scenarios

A behavior is a combination of action sequences to be supported by a system or its components, which can be a function, a software component or a hardware device. A behavior specification provides information about the required or offered behaviors, either during normal or erroneous conditions. Depending on the lifecycle stages of a system, behavior specifications can exist in different levels of abstraction. In general, three categories of behaviors can be identified.

- **Behavior of Computation** – relating to the input and output data handling (e.g., through IO queuing) and the logical transformations of a component/system (e.g., through the mode and the computation thread-of-control). For example, it specifies how the computation operations offered by a component can be invoked either due to the received external signals or because of internal computation needs. Such behaviors need to have access to the component/system interaction points (e.g. ports), internal data (e.g. local variables), and the contained parts (e.g. subprograms).

- **Behavior of Execution&Communication** – relating to the execution of computation behaviors and sending, transmission, and delivery of information between such behaviors under the constraints of precedence and synchronisation. In a multitasking paradigm, the execution of a component or communication behavior is dispatched, either in a time-triggered (periodically) pattern or on an event-triggered (sporadically or aperiodically according to the received data) basis. While a precedence constraint provides information about the causal ordering of behaviour executions, a synchronisation constraint provides information about the dependency of behaviour executions in timeline. Once activated, an execution is constrained by attributes related to timing (such as execution delay, response time, and deadline) as well as performance (such as throughput, bandwidth, and failure rates). A communication behavior specifies for example how a communication message is queued/synchronized for sending and delivering, and how the network transmission is controlled (e.g. arbitration by priorities or collision exclusion by tokens). Such behaviors are supported by communication protocols (e.g., message-passing and CAN devices).

The behavior modeling is needed in system development where precise information in regards to required and offered/allowed application behaviors and execution dynamics is added for different purposes. When associated to a function/component in a system model, a behavior model specifies the actual computation and execution tasks supported by the target artifact. The scope of behavior modeling covers:

- the specification of expected behaviors and behavioral constraints as indicated in requirements and use cases,
- the specification of offered computations of application programs and interface devices,
- the specification of offered logical behaviors of basic software, device managers, and hardware devices,
- the specification of allowed execution dynamics of application programs,
- the specification of related environmental conditions and dynamic behaviors.

In system development, a behavior model can be refined successively with emerging details appearing at the respective abstraction levels in an incremental development process. Through a language support, various behavior specifications can be managed in a consistent and traceable way in multi-views. They constitute the basis for analysis, design and decisions, as well as for ECU executables generation.
5.2 Modelling concepts

5.2.1 Language aspects

A modeling language support behavior modeling normally involve the following aspects

- **Concrete Syntax** – relating to the graphical or textual representation of the language.
- **Abstract Syntax** – relating to the textual representation of the language in a tool for the model maintenance and transformation.
- **Denotational&Axiomatic semantics** – relating to the formalization of the language constructs based on mathematical denotation (e.g., Automata and CSP) or mathematical axiomatic logic (e.g., Hoare logic).
- **Operational Semantics** – relating to the formalization of the language execution as sequences of computational steps, providing the definitions for the operation of a modeling language in a particular platform (e.g. a simulation engine).

5.2.2 Basic models-of-computation in automotive embedded systems

A model-of-computation (MoC) refers to the computation paradigm underlying the execution or simulation of a particular behavior. In effect, a MoC provides the abstract machine on which a behavior model runs. For automotive embedded systems, the following three types of MoCs are most common.

- **Discrete-Time Models** (also referred to as DTSS - Discrete Time System Specification). Discrete-time models specify the state at next time instant given the current state and input using a discrete state transition function using difference equations. They provide a strictly discrete causality of activities of computation such as in feedback loops of automatic control and hence good support to simulations and implementations of continuous time and value models. One fundamental assumption of this MoC is the existence of global synchrony of time.

- **Continuous-Time&Value Models** (also referred to as DESS - Differential Equation System Specification). The continuous-time&value models specify the rate of change on state variables using derivative functions, instead of directly specifying the state at next time instant as in the discrete-time models. The continuous-time&value models provide accurate modeling for many physical systems, such as relating to the dynamics of environmental plant and hardware devices of embedded systems. There are well established techniques for digital simulation of such models, e.g., various numerical integration methods (e.g., Euler, RungeKutta) and solvers have been used for estimating future values of state variables. Feedback loops where a state variable are fed back to as an input to its own derivative or other state variables are supported by strict causality in simulation tools such as through the use of unit delay in terms of integrator.

- **Discrete-Event Models** (also referred to as DEVS - Discrete Event System Specification). In this model of computation, a component/system reacts to events occurring at discrete time points on a time line. An event can be external, caused by the environment, or internal, produced and managed by the system itself. Each component treats the events in chronological order. Discrete-event models have been widely used for modeling software logics and communication behaviors, and for analyzing logic properties like invariants and reachability through model-checking.
For the implementation of embedded EE systems, a widely used model of computation is the multitasking models. This type of models targets the implementation executions of software through multitasking, relating to the timing and scheduling logic of task activation and execution such as the event-triggered and time-triggered schemes. A further model of computation common in embedded EE systems implementation is the imperative semantics as in C programs, concerning the low-level thread-of-control that changes the state of program towards giving the computation result.

Continuous-time systems will primarily be used to model the plant and the environment model, although it could possibly be useful for modeling control functions at a high abstraction level (e.g. continuous PID controllers). As described in I3.4.1, continuous-time systems can be described at different levels of abstraction, letting tools perform translation into lower realization levels. The lower realization levels could be at the same MoC as the embedded system, e.g. Discrete-Time Models, (DTSS), Discrete-Event Models (DEVS), or C programs.

The term “physical modeling” is referring to the method of modeling physical systems by using reusable software simulation components that connects to each other just like the physical object they try to describe. This could then be seen as a domain-specific language for physical systems. The Modelica language use equation-based modeling, together with object-oriented techniques as a means to enable physical modeling.

Many languages and formalisms for behavior modeling have been developed. For example, the synchronous languages Lustre [3] and Esteral [4] are two variants of discrete-time models. These synchronous languages are widely used for avionics and other safety critical system. CCS [5] and CSP [6] are two variants of discrete-event model with rendezvous for synchronizing the state transitions of process. These two process algebra approaches have been widely used for analyzing logical behaviors such as in communication protocols. Compared to other MoCs, the programming language C can also be considered as a low level modeling of software behavior. It is imperative in the sense that the MoC are directly related to the commands for the computer to perform.

5.2.3 Integration of models

In the development of complex embedded systems, the integration of models in different MoCs is necessary for the reasons of system synthesis, analysis and V&V. There are different degrees of integration, ranging from only keeping the inputs, outputs, and time bases in correspondence, to enabling complete transformations between MoCs. There are basically two essential approaches to the integration at model level [1].

I. **IO Matching.** This approach provides encapsulation techniques and mapping semantics so that the inputs and outputs of each pair of MoCs are mapped, allowing the interactions of models by the related MoCs in a meaningful way.

II. **Common semantics.** The other is to provide a common semantics or formalism by which properties of different MoCs can be expressed and managed. This common formalism can then constitute the same generic framework on which the mapping, re-partition, and communication of MoCs are performed.

A common semantics can be obtained by combining and substituting the original ones, or by using one of the MoCs to which other MoCs can be reduced and represented [2]. For example, it is common to reduce all MoCs of concern into an imperative semantics (e.g., C programs), which normally involves a lot of refinement or implementation effort.

With the behavior attributes declared in the language, external formalisms and tools (e.g., Simulink, UML, C code etc.) would be allowed to complement the behavior definitions for analysis, function design, comprehension and communication due to technology or domain
concerns. For example, varying modeling and analysis tools will be used by different automotive companies and suppliers, and in different development stages.

5.3 Methodology

In general, behavior modeling is performed in alignment with the engineering stages within system development and quality design.

As a major task, behavior models can be used to refine textual specifications of requirements. A requirement is mapped to some artifacts for the implementation, derived to other requirements or constraints, and verified by V&V efforts. Meanwhile, a requirement can be refined by behavior models for a precise definition of the semantics and thereby the constraints and its interdependences with other requirements. The related behavior modeling task refines functionality related requirements and provides support for behavior analysis and simulation in external tools (e.g., UML Use Case and Sequence Diagram, State-Machine) for validation and consistency control of requirements.

Another important task of behavior modeling focuses on the logical behaviors of system functions as they transform I/O data, interacts with each other or the system environment. It defines the actual computations to be supported by an E/E system and how the end-to-end computations will be partitioned and allocated to the system functional components. It also derives new requirements and constraints while taking the design of logical behaviors and their mapping to structure design into consideration. Control engineering, as a specialization of this task, carries out the design, analysis and simulation of embedded control functionality (e.g., defining the feedforward and feedback control, state observers and I/O devices, and restrictions on control stability and performance, environment behaviors).

A further behavior modeling task focuses on the integration of logical behaviors of application, basic software, system and hardware platform for fulfilling the technical requirements. The design derives requirements and constraints on software and hardware implementation by taking the resource deployment commitment into consideration.

The behavior design can be performed either before the corresponding structure design tasks (e.g., in a top-down approach) or after the structure design tasks (e.g., when existing solutions are reused). With the specifications of logical behaviors in an architecture description language, external tools (e.g., MATLAB/Simulink) can be incorporated for behavior design, analysis, and simulation.
5.4 Plugin

A MATLAB/Simulink plugin was developed in order to make a reference behavioral model exchange implementation, because it is also a very common tool to use for model-based development of automotive embedded software.

Simulink has an extensive set of tool data using a large number of different element types and parameters. Not all of the details contained in the tool data are relevant for the integration with EAST-ADL. A large intermediate model would make the structural bridge more complicated and harder to maintain. That is why in the meta-model (see Figure 19), we only consider blocks using the model reference mechanism (SystemReference in the figure).

The purpose of the structural bridge is to map concepts between EAST-ADL and Simulink in such a way that the semantics of the original model is preserved. EAST-ADL use the type – prototype mechanism where the declaration is in the EAST-ADL FunctionType, and the usage is in the FunctionPrototype. The FunctionPrototype is additionally contained within a third FunctionType corresponding to the subsystem of a Simulink block. A FunctionType has a pointer to an external behavior, in this case it will be an .mdl file, which is the file format of Simulink models. A FunctionPrototype corresponds to the usage of this .mdl file in a Simulink model using the model reference mechanism. We use ATL for implementing the structural bridge.

<table>
<thead>
<tr>
<th>EAST-ADL Concept</th>
<th>MATLAB/Simulink</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADLFunctionType</td>
<td>.mdl file</td>
</tr>
<tr>
<td>ADLFunctionPrototype</td>
<td>Model reference block</td>
</tr>
<tr>
<td>Connector</td>
<td>Line</td>
</tr>
<tr>
<td>Port</td>
<td>Port</td>
</tr>
</tbody>
</table>

Figure 18: Concept mapping of EAST-ADL and MATLAB/Simulink
Figure 19: The Simulink metamodel used for the transformation
6 References


