# View-Supported Rollout and Evolution of Model-Based ECU Applications

Andreas Polzer
Embedded Software Laboratory
RWTH Aachen University
Aachen, Germany
polzer@cs.rwth-aachen.de

Daniel Merschen
Embedded Software Laboratory
RWTH Aachen University
Aachen, Germany
merschen@cs.rwth-aachen.de

Jacques Thomas
Daimler AG
Group Research & Advanced Engineering
Böblingen, Germany
jacques.thomas@daimler.com

Bernd Hedenetz
Daimler AG
Group Research & Advanced Engineering
Böblingen, Germany
bernd.hedenetz@daimler.com

Goetz Botterweck
Lero – The Irish Software Engineering Research Centre
Limerick, Ireland
goetz.botterweck@lero.ie

Stefan Kowalewski
Embedded Software Laboratory
RWTH Aachen University
Aachen, Germany
kowalewski@cs.rwth-aachen.de

## ABSTRACT

When applying model-based techniques to the engineering of embedded application software, a typical challenge is the complexity of dependencies between application elements. In many situations, e.g., during rollout of products or in the evolution of product lines, the understanding of these dependencies is a key capability. In this paper, we discuss how model-based techniques, in particular, model transformations can help to reduce the complexity of such analysis tasks. To this end, we realised a representation of Simulink models based on the Eclipse Modeling Framework (EMF). The resulting integration allows us to apply various model-based frameworks from the Eclipse ecosystem. On this basis we developed a view that increases the visibility of functional dependencies, which otherwise would have been hidden due to a lack of abstraction in the native Simulink representation. The provided analysis framework comes in handy, when such a model has to be modified. Consequently, the developer is supported in reusing existing models and avoiding errors. The concepts and techniques are illustrated with a running example, which is derived from a real industry model from Automotive Software Engineering.

## Keywords

Model-based development, variability, Matlab Simulink, automotive software, model transformation, ATLAS Transformation Language (ATL), Epsilon Translation Language (ETL)


*MOMPES* '10, September 20, 2010, Antwerp, Belgium


## 1. INTRODUCTION

Model-based development (MBD) of ECU applications has become an established methodology for automotive electronics. The behaviour of the application (e.g., exterior light control, 12V power network management) is first modelled using, e.g., Target Link, then ECU code is automatically generated. Finally, the code is integrated into the runtime environment of the ECU. Nowadays, the challenge is no longer to introduce such methods and techniques, but to manage the application including legacy versions and current product variants over many years. In doing so, one has to consider many different types of artefacts, for instance, the Simulink model, the requirements specification and the corresponding tests.

In the automotive industry MBD is often used in connection with a rollout of an application in different car lines. Although the reuse level across car lines is high, some differences cannot be avoided nevertheless. Moreover, the application is continuously developed, because of enhancements or newly introduced features. Consequently, there is the major challenge of managing consistent artefacts over time and car lines. With each change or new variant, the application developer is confronted with questions like: "Which functions are there already in the application?", "What are the dependencies between them?", "What effect will a change in requirements / a Simulink subsystem have?", "Are there some other requirements/Simulink subsystems concerned?", "Which consequences will result for the corresponding tests?", or "How can we ensure that the artefacts stay consistent despite the change?". Answering such questions today can only be done by an experienced application expert and involves complex searches in a large number of potentially involved artefacts.

To address these challenges, we propose to support development and variability handling using model-driven technologies. In particular, we apply technologies from the Eclipse ecosystem, such as the Eclipse Modelling Framework (EMF) and corresponding model transformation languages. The

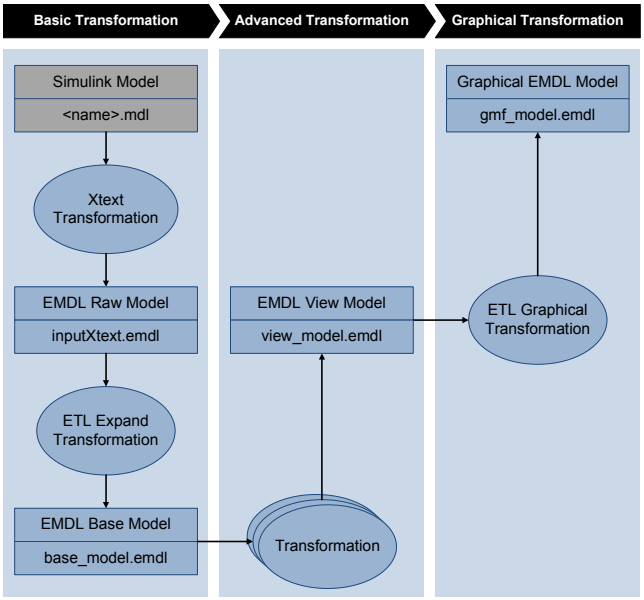


**Figure 1: Overview of the transformation concept**

main idea is to import original artefacts (as used in MBD industry practices) into an EMF-based representation and subsequently apply techniques, which are available in Eclipse-based frameworks. For instance, with such techniques an artefact can be analysed directly or combined with other artefacts for cross-artefacts analyses.

Out of the many involved artefacts, in this paper we focus on the Simulink model. We will first describe the model transformation concept, then we provide an analysis example producing another view of the model applying this concept. Finally we evaluate the concept in the context of benefit for industries.

## 2. TRANSFORMATION CONCEPT

To answer the questions raised in Section 1 we have developed a transformation concept, which is visualised in Figure 1.

The transformation concept consists of three main steps each of which is dedicated to a specific task in the process. First of all the Simulink model is imported into the EMF world during the basic transformation step and enriched with additional logical information. This step is described in more detail in Section 2.1. The result is a model that we call *EMDL Base Model* as we use it as input for further advanced transformations (cf. Section 2.2). These advanced transformations lead to models that represent special task-specific views of the original Simulink model. Such a "view model" can subsequently be transformed to a "GMF model" during the graphical transformation process described in Section 2.3 to be visualised.

### 2.1 Basic transformation

To explain the benefits of the basic transformation step, we will focus on lines that are connecting blocks in a Simulink model as they are the central element to identify those blocks which are transporting a signal and are potentially affected by the signal. For instance, in the very simple Simulink model visualised in Figure 2 we might be interested in which blocks are affected if the constant signal of block "Constant 2" changes. To automate this task we have to analyse the Simulink model text file, which includes all information about the different blocks, their connections, signal flows, graphical positions and so on.

If we want to perform such analysis, the original format of the text file is problematic as an input, because some of the necessary information mentioned above is available only in an implicit form. That is the reason why the task of identifying lines which are connected to block `Constant 2` in Figure 2 becomes quite complicated as illustrated in the following. Each line can be connected either directly to a block or to a port of a block. Lines and blocks are independent objects which are represented in the model file with a keyword ("Line" / "Block") followed by a body surrounded by curly braces "{}". All entities are just listed in a linear order without a logical or hierarchical structure. Listing 1 illustrates how a line between the Simulink blocks `Constant 2`, `Module A`, `Module C` and `Output Interface` of the Simulink model in Figure 2 is represented in the original Simulink model file.

```
1  Line {
2    [...]
3    SrcBlock          "Constant 2"
4    SrcPort           1
5    [...]
6    Branch {
7      [...]
8      Branch {
9        DstBlock      "Module C"
10       DstPort       1
11     }
12     Branch {
13       [...]
14       DstBlock      "Output Interface"
15       DstPort       4
16     }
17   }
18   Branch {
19     DstBlock        "Module A"
20     DstPort         1
21   }
22 }
```

**Listing 1: A line as represented in the original Simulink file**

Answering the previously mentioned question is problematic with this structure. For example, if we would like to retrieve all blocks that are linked by lines to the block `Constant 2` directly or indirectly via further blocks (which is generally a frequent task when identifying blocks affected by a signal) we first have to pick this block in the model file and retrieve its name (here `Constant 2`). Then, we have to iterate through all lines beginning at this block (i.e., which have an attribute "SrcBlock" with this name as value).

One of these lines is displayed in Listing 1. The names of the blocks which are directly connected to `Constant 2` will then be all *DstBlock* entries which appear in the line body of these lines. In order to identify blocks connected with block `Constant 2` indirectly we would have to look for the names of the destination blocks (DstBlock) of all lines which the condition `SrcBlock="Constant 2"` holds for. After this we have to repeat the whole process recursively on the resulting destination blocks.

In summary, the flat structure and textual storage format of original Simulink files, impedes efficient traversal and analysis of the model. This is not a suitable basis when

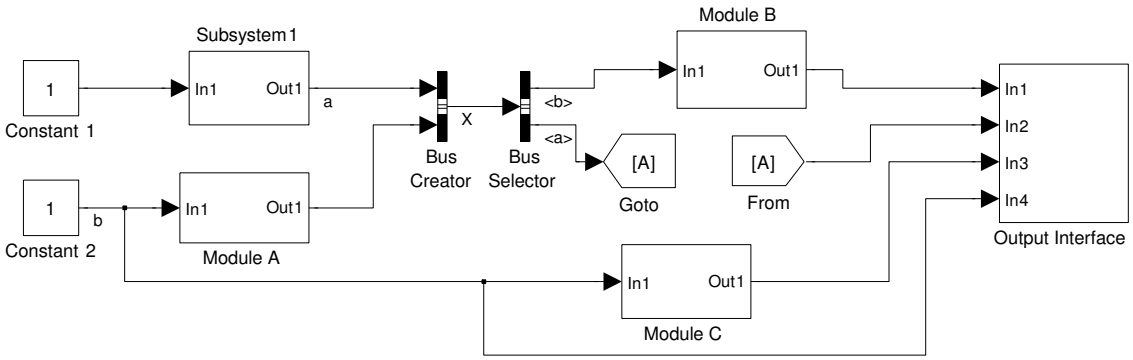


**Figure 2: A Simulink model to be analysed**

aiming to understand and analyse an ECU application, even if such analysis is performed with tool support.

To overcome this obstacle, we decided to explore techniques that would turn implicit information into explicit objects, such that queries on the Simulink model can be implemented and performed more elegantly and efficiently. In order to do so, we created a parser within the Xtext framework [10]. This parser turns a given Simulink model file in the original `.mdl` format into an EMF-based representation, which is stored in XMI (XML Metadata Interchange), a common interchange format for EMF models. This document conforms to an EMDL metamodel, which we defined to capture all information given by the Simulink model. Furthermore, we extended this metamodel by additional logical and hierarchical information, which facilitates analyses. Connections between blocks and lines, for example, will then be easier to address.

The Xtext parser is started via an Xtext workflow script.For the given Simulink model line in Listing 1 it generates the output shown in Listing 2.

```
<mdlLines name="b">
  [...]
  <branches>
    <branches>
      <destinationPointer = "Module C"/>
    </branches>
    <branches>
      <destinationPointer = "Output Interface"/>
      [...]
    </branches>
    [...]
  </branches>
  <branches>
    <destinationPointer = "Module A"/>
  </branches>
  <sourcePointer = "Constant 2">
</mdlLines>
```

**Listing 2: A line as represented after Xtext transformation**

Up to this point we have just translated the original Simulink model file *one-to-one* into the new EMDL Raw Model file, which conforms to the defined meta model. As we mentioned before some important information is implicit in the Simulink model and cannot be addressed directly. This information is still implicit in this EMDL Raw Model. For analyses purposes it is desirable to abstract from technical details to make the required information explicit. For

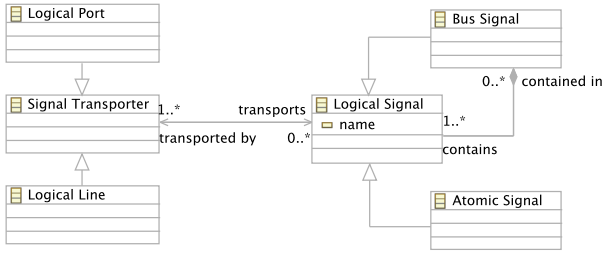


**Figure 3: The structure of logical entities which facilitate the transformation process**

the line displayed in Listing 2 an abstraction would not use the quite technical construct of branches. Instead it would just describe a *logical* line with one source and many targets (which would be references to the corresponding target blocks reached by this signal). Therefore we identified the following logical entities which become relevant in this context and which are created during a further transformation (ETL Expand Transformation):

1. Logical ports
2. Logical lines
3. Logical signals
4. Signal transporters

Figure 3 illustrates a part of the metamodel describing the relationships between these logical entities.

### 2.1.1 *Logical ports*

Each line in a Simulink model is connected either directly to blocks or to ports of blocks. In our enhanced model we facilitate this by introducing logical ports which belong to a block. Logical lines can then only be connected to logical ports. So in this context we do not have to distinguish between blocks and ports as endpoints of lines in a Simulink model. A logical port can either be a logical inport or a logical outport depending on whether a port is a line's target or source.

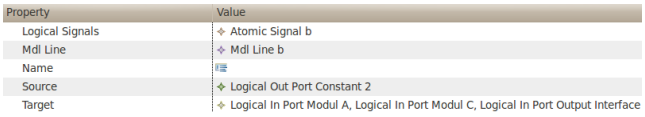


**Figure 4: A logical line that abstracts from details of a Simulink model line**

#### 2.1.2 *Logical lines*

As mentioned before, in the EMDL Raw Model connecting lines between blocks are still represented in a quite technical manner. In other words, each line has exactly one source and one explicit target but it can contain many branches where further lines branch off (see Listing 2). This construct of branches is not necessary as it does not introduce more semantical information into the model useful for analysing signal flow.

For analysis purposes we are mostly interested in what block (or port respectively) is connected with which other blocks and which signals are transported. Hence, we introduced the logical lines which abstract from the details of *how* a line is connected with different blocks. Instead a line has exactly one logical source port and at least one logical target port but no branches as they do not introduce relevant information about signal flow. Consequently, there are no branching lines left which facilitates the process of analysing connections and signal flow between blocks significantly.

Figure 4 illustrates the logical representation of the Simulink line of Listing 1 (and MdlLine of the EMDL Raw Model of Listing 2). Furthermore, even visually separated blocks like `from` and `goto` blocks in the Simulink model are connected by a logical line.

#### 2.1.3 *Logical signals*

While information about the signal flow is contained implicitly in the original Simulink model as described above (and hence difficult to extract) we create logical signals, which have a name and are related to all ports and lines transporting the signal. For example, in order to identify the signal flow of a given signal in the Simulink model we have to follow the lines beginning at the source block. But a line may also transport more than one single signal, i.e., it transports a bus signal that contains all transported signals.

To this end, we also introduce logical signals that can either be bus signals, which contain one or many further logical signals or atomic signals that may be contained in a bus signal but cannot transport other signals. Furthermore, each atomic signal is linked to all bus signals, which transport it. By this means, it becomes apparent for a given signal which lines transport this signal (either included in a bus signal or directly). For a given atomic signal we can now directly lookup which line is associated (either to this atomic signal or to a bus signal that contains this signal). Now we can perform queries on the model to identify parts of a model which are potentially influenced when a signal changes.

#### 2.1.4 *Signal transporters*

In a Simulink model signals are transported between blocks. In order to facilitate identifying signal flows throughout a Simulink model both logical lines and logical ports are abstracted as “signal transporters”. By having such an abstraction at hand the problem of finding out, which parts of a model are influenced by a special signal, becomes solvable with a reasonable effort since the information about signal flow now becomes explicit. That means that we no longer need to follow a given signal through a Simulink model. Instead, we just have to query our transformed model for relevant signal transporters for the signal under consideration.

### 2.2 Advanced transformations

By enriching the original Simulink model with abstract logical information we created a useful base model, which now allows us to answer questions about the ECU application and the corresponding Simulink model with reasonable effort (e.g., ”Identify all lines and blocks of the model which are influenced by a change of signal ’a’.”). The second step of the concept is then to extract some information out of the EMDL Base Model and process it further. For that step we use transformations that we call *Advanced Transformations*. One of these is shown in Section 3. In future extensions these advanced transformations may also consider further input as exemplified in Section 6. The result of this step is called the *view model*, which represents those aspects that are relevant to a particular stakeholder or for a particular task.

### 2.3 Graphical transformation

To further support the engineer in his work, one objective is to provide tools, which provide interactive access to the task-relevant information. Hence, we aimed to produce a graphical representation of the view model to make the result of the analysis more comprehensible and intuitive. To this end, we have developed a graphical editor using Eclipse GMF (Graphical Modeling Framework).

The editor enables the user to recursively descend into subsystems by double-clicking them – similar to the behaviour known from Simulink. The graphical representation uses available positioning information as given by the Simulink model to arrange the blocks on the canvas.

Each model to be visualised in the editor has to be structured such that the editor can display it. Therefore, our transformed model has to be enhanced such that it provides the necessary information. This is the task of the graphical transformation. Especially lines and – if desired – branches of lines are introduced within this transformation. Now the created view can be visualised. The result of the graphical transformation is called a *GMF model*. Section 3 applies the described methodology to create such a graphical editor view.

### 2.4 Transformation languages

For all transformations we applied two state-of-the-art transformation languages, selecting one of them depending on the adequacy for a given problem. While the ATLAS Transformation Language (ATL) [8] succeeds in transforming huge models very efficiently, the Epsilon Translation Language (ETL) [9] seems to be more convenient to apply for a developer used to programming with Java or C++.

This is why we mainly use ETL whenever concrete modifications on a model have to be performed (e.g., to create special view in the advanced transformations) whereas we apply ATL to add minor additional information on possibly large models (e.g., to create logical signals and to distinguish bus and atomic signals). With transformation languages it is difficult to directly manipulate a given model in place (similar to accessing and modify objects in an object-oriented

programming language). Hence, we created what we call "identical transformations" both for ETL and for ATL. These transformations create an output model that is identical to the input model. That way, we can build new advanced transformation just by modifying these identical transformations as necessary.

It should be noted that the applied transformation languages (both ATL and ETL) provide some similar support for copy-and-modify transformations, e.g., in ATL's refine mechanism or in ETL's capability to generate a copy transformation. However, we had mixed results with these and are still exploring our options to find the best solution for use in everyday practice.

# 3. ANALYSIS OF SIMULINK MODELS

The previous section explained the transformation concept. Such transformation of a Simulink model into the Eclipse world enables to use all techniques available from the Eclipse Modeling Project, e.g., corresponding transformation languages. In this section we apply these concepts to support the development of Simulink models. In particular, we are aiming to support a development strategy, where a model is built out functional, compatible units taken from a library. We will refer to this approach as "module-based".

A Simulink model is often used to generate code, for instance with Target Link. As models are used by different partners many standardisation techniques have to be adopted (e.g., to comply with AUTOSAR [4]). For example abstraction techniques have to be adopted to follow special design patterns. Parts of the whole system might be grouped into subsystems, which in turn themselves may contain subsystems. Furthermore, signals may be integrated into buses. Consequently, the structure of a Simulink model often becomes very complex.

Especially, as we are dealing with families of products (e.g., similar implementations for various types cars) the implementation model contains the functionality for a whole family of variants. These variants are used to adopt the models for different vehicles where slightly different changes are necessary. Therefore different patterns are used to enable variant management of the functionality. In earlier work, we presented some approaches to manage and configure variants [12, 11]. One of these patterns is a subsystem called `Module`, which encapsulates the functionality of one feature.

Figure 5 shows the pattern of a module. There are different activation states. The *Disabled* state is set when the feature/module is never active in the variant (e.g., due to an functional option that is not available in this particular car type). When the module is *Enabled* it is either active or inactive. *Active* means that the module is running while an *Inactive* module is available but not running. The different activation states are controlled by the subsystem **A**.

With this structure the set of all modules can be divided into different groups of running modules. Again we have to deal with the complexity of the system, which is hard to handle for the human engineer. The modules running at a specific car state are not easy to grasp and understand by a developer, neither on the requirement specification level nor on the level of the Simulink model implementing these requirements. To analyse the model the developer needs to "see" which module is active at which state and which dependencies exist between modules.

As an example for the above mentioned issue we assume

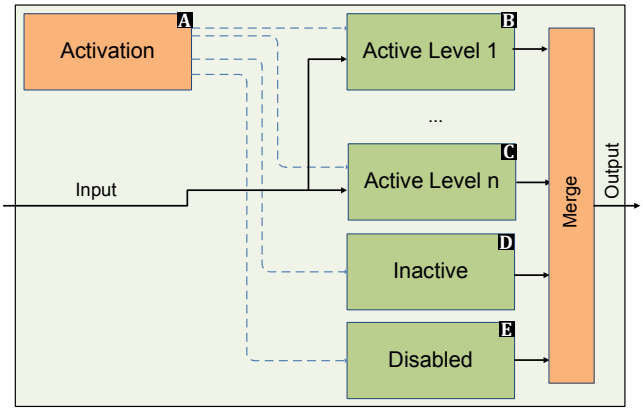


**Figure 5: Structure of the module pattern**

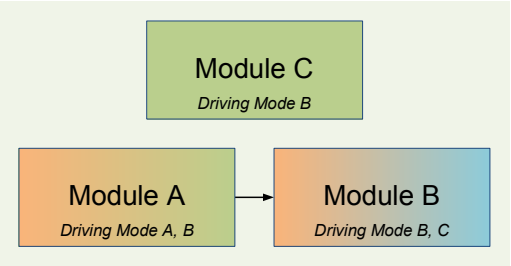


**Figure 6: Example for a module structure which might cause problems**

the Simulink model given in Section 2 shown in Figure 6. The `Module A` is activated in `DrivingMode_a` and `DrivingMode_b` and providing signals to `Module B`. There is no problem with `Module C` which is activated in `DrivingMode_b` does not depend on signals from `Module A`. But `Module C` is activated in `DrivingMode_B` and `DrivingMode_C`. So it expects to have input from `Module A` in both modi. But `Module A` does not provide any signals in `DrivingMode_C`. This might cause an error.

One important task when developing the Simulink model is the analysis of dependencies of modules within the models. Hence, our work aims at providing a view which shows the dependencies of modules and the states where the modules are activated. The view should depict all modules on one pane with arrows indicating the dependencies and the structure which shows the activation.

To do so, we first import the Simulink model into EMF as described in Section 2.1, then in a second step we have to transform the EMDL Base Model to the desired "view model", showing the dependencies between modules. In the next chapter we describe this advanced transformation.

## 3.1 Transformation of the Simulink Model

We have written two advanced model transformations (see the structure shown in Figure 7), which identify the modules at a first stage ❶ and transform ❷ the modules to a structure according to the requirements described above.

The transformation identifying modules in the Simulink model is written in the ATLAS Transformation Language

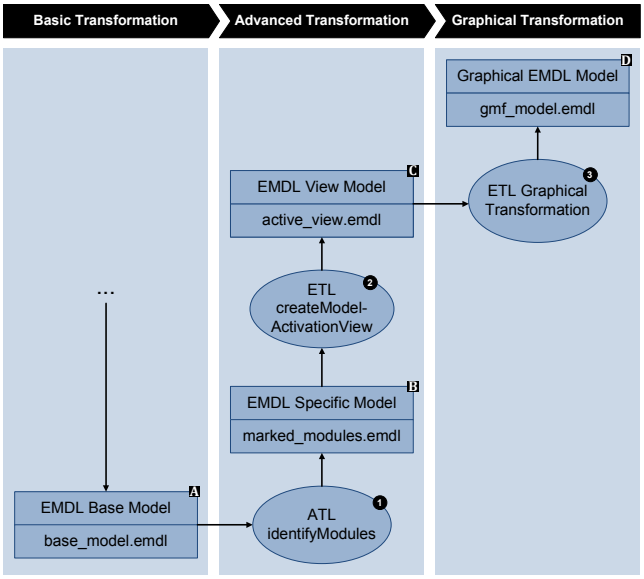


**Figure 7: Overview of the model transformation to create a flattened module structure**

(ATL) [8]. Modules can be identified via a tag added to the Simulink model. Each found module is transformed into an instance of "Module" (a metaclass introduced in the metamodel). By these means it is possible to identify the modules in later processing steps.

The second transformation creates a new model consisting only of modules and dependencies. To do this, we used transformation rules written in Epsilon Transformation Language (ETL) [9]. Listing 3 describes the essential rule for a root system and the computation for new targets.

The rule `CreateRootSystem` converts a normal `System` (input model) to a new special system (output model) where only blocks of the type `Module` are available. The transformation of these new blocks can be found in lines 13-18, where a loop transforms all instances of Module in line 15 and adds them to the target model in line 17.

Within this rule we also adapt the lines such that they should now indicate, that there is a connection between modules. These connections are created by the operation `createNewLine` which is called in line 19. Due to space constraints we do not depict this function. Its main contribution is to create a new line with the source module $m$ from the loop and the targets which are computed by the operation `getLinkedModules`.

```
1  [...]
2  rule CreateRootSystem
3    transform    s:MdlIn!System
4    to           t:MdlOut!System
5    extends NamedElement
6    {
7
8      var allModules :new Set(MdlIn!Module);
9
10     for (m in MdlIn!Module.allInstances()){
11       allModules.add(m);
12     }
13
14     // add only module blocks
15     for (m in allModules)
16     {
17       var outModule : MdlOut!Module = m.equivalent
                ("CreateModule");
18       t.blocks.add(outModule);
19
20       //create the structural connection
21       t.logicalLines.add(createNewLine(m, m.
             getLinkedModules(allModules));
22     }
23   }
24   [...]
```

**Listing 3: Transformation rule for System given in ETL**

The operation mentioned above, which searches for linked modules is an important operation, which is depicted in Listing 4. This operation is realized as a recursive object-oriented function. For a given block and a set of `Subsystems` given as parameter it returns a subset of `Subsystems` which are connected to the calling block. The result will be found in the variable `linkedModules`.

As illustrated in line 3 the operation therefore iterates through all outports of the block which the operation is invoked for. If the output is linked to a logical line we retrieve all targets of the line (line 5-7) and decide what to do with them based on the block type of the target block of the line.

If the target is contained in the set of our modules we have found a connection and add it to our result (lines 8-10). If we have found another subsystem we have to search for modules within this subsystem. So we are calling this function again recursively (line 12) and add all results we obtain from this call. If we have reached the end of a subsystem we leave it and continue searching in the containing system (line 14). For all other types of blocks we call the function again recursively (line 16).

```
1  operation MdlIn!Block getLinkedModules(modules :
       Set(MdlIn!SubSystem)) : Set (MdlIn!SubSystem)
       {
2    var linkedModules : new Set(MdlIn!SubSystem);
3
4    for (outPort in self.outPorts){
5
6      //  is there a Logical Line
7      if (outPort.linkingLine.isDefined()){
8
9      // Search in all Targets
10       for (target in outPort.linkingLine.target){
11
12         if (modules.includes(target.parent)){
13           linkedModules.add(target.parent);
14         }
15         else if(target.parent.isTypeOf(MdlIn!
               SubSystem)){
16       linkedModules.addAll(target.accordingBlock.
             getLinkedModules(modules));
17         }
18         else if (target.parent.isTypeOf(MdlIn!
               BlockOutPort)){
19           linkedModules.addAll(target.
                accordingPort.parent.
                getLinkedModules(modules));
20         }
21          else {
22           linkedModules.addAll(target.parent.
                getLinkedModules(modules));
23         }
24       }
25     }
26   }
27   return linkedModules;
28 }
```

**Listing 4: Operation which determines the targets for a given module**

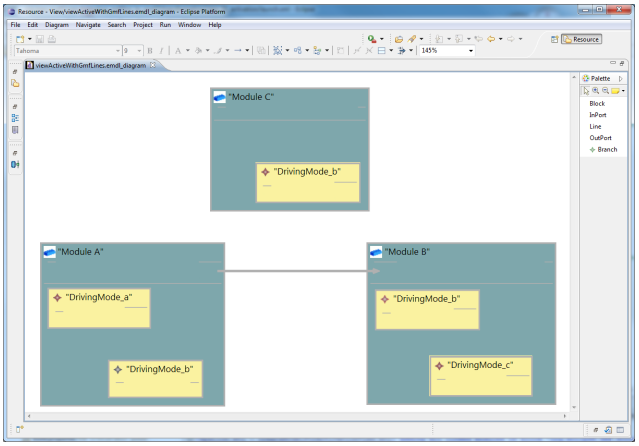


**Figure 8: Graphical editor for the transformed Simulink Model**

With the algorithm in Listing 4 we are able to scan the whole model structure no matter if there are `subsystems`, `bus structures`, `goto` or `from` blocks. Hence we are able to find the modules that are connected with a special module very easily due to the fact, that we did a lot of structural work before.

The resulting model now contains the structure we are searching for. All modules have been lifted to the same layer and connected if there is a connection in the source model.

## 3.2 Graphical Editor

In Figure 8 we show how the transformed model appears in the editor.

We used the example from Figure 2 in Section 2. After the transformations the modules A, B and C are on one layer and if modules are interconnected this it depicted by lines, e. g., like between module A and module B. Within the modules the developer can see the structural content. Therefore he is able to decide in which levels the module is activated.

# 4. EXPERIENCES AND EVALUATION OF VIEWS

We applied the presented transformation concept on real examples from automotive industry. These examples are Simulink models which have up to ten different *Driving Modes* similar to the presented example in Section 3. With this concept module dependencies and their activation can now automatically be extracted out of the Simulink model. The application developer no longer has to do the exhausting job of looking for this information in the Simulink model.

The views allow to save time and to avoid losing track during search in the model. Additionally the graphical representation facilitates the analysis of the information. Adding a new module in the Simulink model or changing an existing one is now better supported.

During early testing we noticed that the presented transformations work correctly and efficiently when applied on less complex Simulink models. However, there seem to be enormous differences between the used transformation languages ETL and ATL. For instance, by (manual) translation of an ETL transformation into an equivalent ATL transformation we could increase the performance of the transformation by factor 60 related to one and the same Simulink model. It is too early to give detailed report on the reasons for this discrepancy, but it surely seems to be worth further investigations.

Furthermore, we discovered that independent of the transformation language very complex models caused heap space errors on a normal PC (AMD Dualcore with 3 GB RAM). This currently stops us from performing the presented transformation with reasonable effort for very complex models. We plan to tackle this challenge by (1) translating current ETL transformations into equivalent ATL transformations and (2) shrinking the Eclipse representation of the Simulink models.

The latter means for example to drop out information about the Simulink model which is not necessary for a developer to visualise in a view, e.g., options associated to blocks, technical lines (cf. Section 2.1) and graphical positioning information. However, this means that a transformation of the Eclipse representation back into a Simulink model will no longer be possible. But as we primarily aim at generating views this will be acceptable for the current usage scenario.

# 5. RELATED WORK

Similar to our approach of analysing Simulink models there is the MOFLON framework [3], which provides tools to access Simulink models and other process artefacts. The authors present a method of keeping requirements and Simulink models consistent with the help of model transformations given in the MOFLON framework.

Alhawash et al. [2] provide another framework to develop and analyse automotive software systems. The framework supports the development in an early design phase with different views based on a common model provided by this framework. The concrete specification and implementation is done with Matlab/Simulink.

Agrawal et al. [1] focus on model transformation of Matlab Simulink and Stateflow models in a verification context. Therefore they adopted the Graph Rewriting and Transformation language (GReAT) to build hybrid automata from given models specified in Hybrid Systems Interchange Format (HSIF). In contrast of our work the transformation focussed on the semantics of the Simulink model.

A further important focus on model transformation is presented by Biehl et al. [5]. They use the above mentioned ATLAS Transformation Language (ATL) to automate translations from the automotive architecture description language EAST-ADL2 to a safety analysis tool called HiP-HOPS in the context of model-based development of safety-related embedded systems.

This paper primarily aims to support the developer to manage variability and changes of products or product lines respectively based on the analysis of Simulink models via Eclipse frameworks. Another approach presented in earlier work [12] is managing variability via improving the Rapid Control Prototyping engineering process with the help of formal feature models.

[7] attempts to integrate product configuration and variability resolving into domain specific languages with special focus on dependencies between elements and features. To this end, the authors adopt higher-order transformation languages like the ATLAS transformation language (ATL).

[6] deals with possibilities to translate domain specific languages into configurable models with formal semantics. The presented framework enables a tool-supported configuration process by visualising mapping between features and implementation including explanations about constraining dependencies (e.g., excludes and requires) in a given feature selection. Furthermore they show how negative variability combined with subsequent pruning can be used to derive a product-specific model for a given configuration. Finally they evaluate the presented approach on a parking assistant.

[11] elaborates on challenges which result from combining embedded software products to a product line in model-based engineering and how the upcoming problems can be tackled. These challenges range from complexity handling to tool integration. Therefore they create a Simulink model from a feature model and transform this model into a domain model in the Eclipse world using the Xtext framework. A product can then be derived by selecting the desired features in the feature model and mapping this selection to the previously created domain model.

## 6. CONCLUSIONS

In this paper we presented an approach to analyse the structure of a Simulink model to facilitate the engineering process, in particular with respect to change and variability management. We exemplified the methodology by a model-based developed ECU application inspired by real projects in automotive industry. Hence, our work is based on Simulink models where we identified so-called modules (i.e., subsystems that represent a special feature) and restructured them to enable dependency analyses.

To this end, we created a view using model transformations to determine the interesting blocks and interrelations between them. The visualisation is implemented by a model-based developed editor. Finally, we evaluated our approach on a real industry example.

Generating this particular view was only a first step in evaluating EMF to support the further development of ECU applications. We plan to implement and evaluate diverse further views with respect to their impact on industry development. Moreover, we intend to develop a method or a language suited to define arbitrary special view.

Additionally, we expect benefits and possibilities from integrating further input into such transformations. Integrating artefacts of different engineering phases, e.g., requirements and test cases will enable analyses that support a traceable engineering process of ECU applications. Even formal feature models could be used as further input. For instance, an analysis tools could identify and visualise which parts of the Simulink model will be affected by feature selection.

## 7. REFERENCES


[1] A. Agrawal, G. Simon, and G. Karsai. Semantic translation of simulink/stateflow models to hybrid automata using graph transformations. *Electronic Notes in Theoretical Computer Science*, 109:43–56, 2004. Proceedings of the Workshop on Graph Transformation and Visual Modelling Techniques (GT-VMT 2004).

[2] K. Alhawash, T. Ceylan, T. Eckardt, M. Fazal-Baqaie, J. Greenyer, C. Heinzemann, S. Henkler, R. Ristov, D. Travkin, and C. Yalcin. The fujaba automotive tool suite. In *Proc. of the 6th International Fujaba Days 2008, Dresden, Germany*, 2008.

[3] C. Amelunxen, A. Königs, T. Rötschke, and A. Schürr. MOFLON: A Standard-Compliant Metamodeling Framework with Graph Transformations. In A. Rensink and J. Warmer, editors, *Model Driven Architecture - Foundations and Applications: Second European Conference*, volume 4066 of *Lecture Notes in Computer Science (LNCS)*, pages 361–375, Heidelberg, 2006. Springer Verlag, Springer Verlag.

[4] AUTOSAR. Autosar - automotive open system architecture. http://www.autosar.org.

[5] M. Biehl, C. DeJiu, and M. Törngren. Integrating safety analysis into the model-based development toolchain of automotive embedded systems. In *LCTES '10: Proceedings of the ACM SIGPLAN/SIGBED 2010 conference on Languages, compilers, and tools for embedded systems*, pages 125–132, New York, NY, USA, 2010. ACM.

[6] G. Botterweck, A. Polzer, and S. Kowalewski. Interactive configuration of embedded systems product lines. In *Proceedings of the 1st International Workshop on Model-driven Approaches in Software Product Line Engineering(MAPLE 2009), collocated with the 13th International Software Product Line Conference (SPLC 2009)*, volume 557, pages 29 – 35, San Francisco, California, USA, August 2009. CEUR Workshop Proceedings. ISSN 1613-0073.

[7] G. Botterweck, A. Polzer, and S. Kowalewski. Using higher-order transformations to derive variability mechanism for embedded systems. In *2nd International Workshop on Model Based Architecting and Construction of Embedded Systems (ACESMB 2009), Workshop at the 12th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2009)*, Denver, Colorado, USA, September 2009.

[8] Eclipse-Foundation. Atl (ATLAS Transformation Language). http://www.eclipse.org/m2m/atl/.

[9] Eclipse-Foundation. Epsilon. http://www.eclipse.org/gmt/epsilon/.

[10] Eclipse-Foundation. Xtext - a programming language framework. http://www.eclipse.org/Xtext/.

[11] A. Polzer, G. Botterweck, I. Wangerin, and S. Kowalewski. Variabilität im modellbasierten Engineering von eingebetteten Systemen. In *7. Workshop Automotive Software Engineering*, volume P-154 of *Lecture Notes in Informatics (LNI)*, pages 2702 – 2719. Gesellschaft für Informatik (GI), 2009.

[12] A. Polzer, S. Kowalewski, and G. Botterweck. Applying software product line techniques in model-based embedded systems engineering. In *Model-based Methodologies for Pervasive and Embedded Software (MOMPES 2009), Workshop at the 31st International Conference on Software Engineering (ICSE 2009)*, pages 2–10. IEEE Computer Societ, May 2009.