# Model Transformations for Migrating Legacy Models: An Industrial Case Study

Gehan M. K. Selim[1], Shige Wang[2], James R. Cordy[1] and Juergen Dingel[1]

[1] School of Computing, Queen's University, Kingston, Ontario, Canada, K7L3N6
[2] Electrical and Controls Integration Lab, General Motors Research & Development, Warren, Michigan, USA, 48090
gehan@cs.queensu.ca, shige.wang@gm.com, {cordy, dingel}@cs.queensu.ca

**Abstract.** Many companies in the automotive industry have adopted MDD in their vehicle control software development. As a major automotive company, General Motors has been using a custom-built, domain-specific modeling language, implemented as an internal proprietary metamodel, to meet the modeling needs in its control software development. As AUTOSAR (AUTomotive Open System ARchitecture) is being developed as a standard to ease the process of integrating components provided by different suppliers and manufacturers, there is a growing demand to migrate these GM-specific, legacy models to AUTOSAR models. Given that AUTOSAR defines its own metamodel for various system artifacts in automotive software development, we explore using model transformations to address the challenges in migrating GM legacy models to their AUTOSAR equivalents. As a case study, we have built a model transformation using the MDWorkbench tool and the Atlas Transformation Language (ATL). This paper reports on the case study, makes observations based on our experience to assist in the development of similar types of transformations, and provides recommendations for further research.

**Keywords:** Model Driven Development (MDD), model transformations, AUTOSAR, transformation languages and tools, automotive control software

## 1 Introduction

MDD is a relatively new software development methodology that uses models for software specification and communication. In MDD, software development is a sequence of model transformations where abstract models are successively converted into detailed models, and eventually into code. Model transformations are implemented using a model transformation language, which can be declarative, imperative, or hybrid. While a declarative language yields a compact specification, an imperative language is more capable of specifying complex transformations.

As one of the early MDD adopters in industry, General Motors (GM) has created a domain-specific modeling language, implemented as an internal proprietary metamodel, for Vehicle Control Software (VCS) development. The metamodel defines modeling constructs for vehicle control software development, including schedules

and interfaces. VCS models conforming to this metamodel have been used in several vehicle production domains at GM, such as body control and monitoring.

Recently, AUTOSAR (the AUTomotive Open System ARchitecture) [2] has been developed as an industry standard to facilitate integration of software components from different manufacturers and suppliers and enable exchangeability and interoperability among them. AUTOSAR defines its own metamodel with a well-defined layered architecture and interfaces. Since converging to AUTOSAR is a strategic direction for future modeling activities, transforming GM legacy models to their equivalent AUTOSAR models becomes essential. Model transformation is a key enabling technology to achieve this convergence objective.

Despite the existence of studies in MDD industry adoption [19][23], no transformation is reported to have migrated legacy models in the automotive industry. To test the practicality of using transformations for migrating industrial legacy models, we have implemented a transformation of GM legacy models to AUTOSAR models.

The rest of this paper is organized as follows. Section 2 discusses the process context in which our transformation is implemented. Section 3 describes the source and target metamodels of the transformation. Section 4 details the transformation development. Section 5 discusses our experiences and issues that require further research. Section 6 provides a summary, a comparison to related work and future work.

## 2 VCS Development, Models and Model Transformations

Applying transformation requires understanding of the development process, which provides a context for the transformation. The VCS development process is described as a V-diagram (Fig. 1). The stages on the left-hand side of the V-diagram are design and implementation activities, and the stages of the right-hand are integration and validation activities. The design starts from system requirements models, which are decomposed into hardware and software subsystem requirements models. The subsystem requirements models then are assigned to engineering groups for refinement into design models and then implemented by hardware and software components. These implemented components are integrated into Electronic Control Units (ECUs), configured for a designated vehicle product. The components are then tested at various levels against their models on the same level on the left-hand side of the V-diagram.

Different types of models in different formalisms are manipulated in the VCS development process. For example, control models use differential equations and timing-variation functions; software models use dataflow diagrams or class diagrams; and architecture models use annotated block diagrams. Selected modeling tools (e.g., Simulink, Rhapsody) and languages (e.g., UML, AADL) are used for modeling.

The transformations used in the VCS development process can be *horizontal or vertical transformations*. Horizontal transformations manipulate models at the same abstraction level but possibly in different formalisms, e.g. transforming a Matlab Stateflow state machine into a UML state machine. Such transformations are normally used to verify integration of subsystems to realize a system function. The source and target modeling languages may have different syntax, but must share similar se-

mantics. Vertical transformations manipulate models at different abstraction levels, e.g. generating a deployment model from software and hardware architecture models. Vertical transformations are usually more complex than horizontal transformations due to the different semantics of the source and target models.

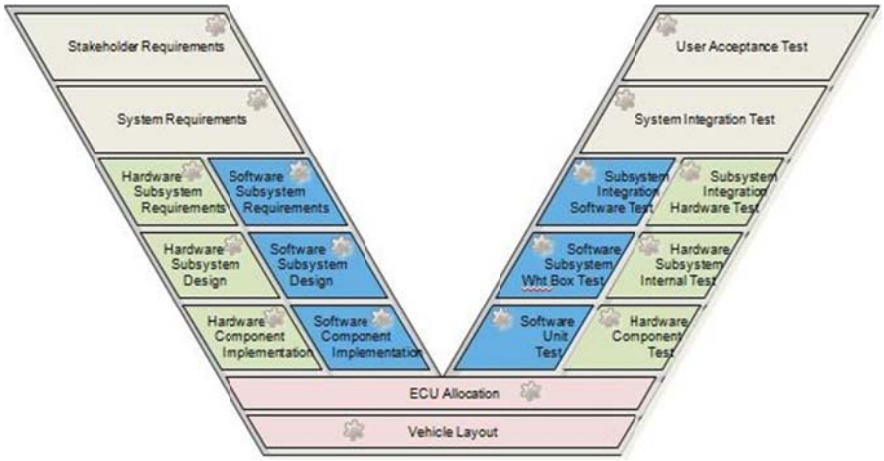



Fig. 1. V-Diagram for the VCS development process.

## 3 Source and Target Metamodels

In this study, our models are those generated and used at the software subsystem design stage in the VCS development process. The source metamodel is an internal, proprietary GM metamodel which we will refer to as the GM metamodel. The target metamodel is the AUTOSAR System Template [2]. To simplify the exercise without losing generality, a subset of the two metamodels is manipulated in the transformation. Specifically, we focus on the modeling elements related to the software components' deployment and interactions, as discussed below.

### *3.1* The GM Metamodel

Fig. 2 illustrates the meta-types in the GM metamodel[1] that represent the physical nodes, deployed software components and their interactions. The *PhysicalNode* type specifies a physical node on which software is deployed. A *PhysicalNode* may contain multiple *Partition* instances, each of which defines a processing unit or a memory partition on which software is deployed. Multiple *Module* instances can be deployed on a single *Partition*. The *Module* type is the atomic, reusable element in a product line and can contain multiple *Scheduler* instances. The *Scheduler* type is the basic

[1] The metamodel has been altered for reasons of confidentiality. However, the relevant aspects required for the purpose of this paper have all been preserved.

unit for software scheduling and manages services provided or required by behavior-encapsulating entities. Thus, each *Scheduler* may provide or require many *Service*s.

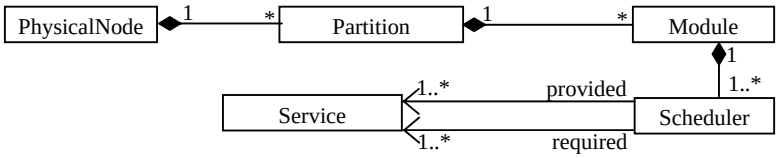



Fig. 2. The subset of the GM metamodel used in our transformation.

### *3.2* The AUTOSAR Metamodel

The AUTOSAR metamodel is defined as a set of templates, each of which is a collection of classes used to specify an AUTOSAR artifact. The *System* template [3] is used to capture the configuration of a system or an Electronic Component Unit (ECU). An ECU is a physical unit on which software is deployed. When used for the configuration of an ECU, the template is referred to as the *ECU Extract*. Fig. 3. shows the metatypes in the ECU Extract that capture software deployment on an ECU.

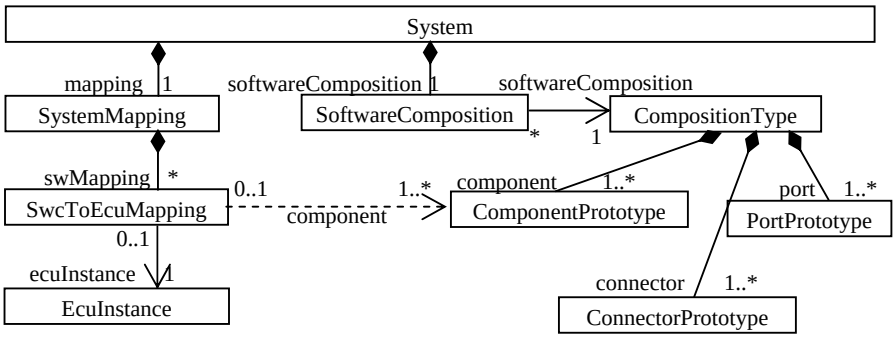



Fig. 3. The AUTOSAR System Template containing relevant types used by our transformation.

The ECU extract contains the *System* type which aggregates *SoftwareComposition* and *SystemMapping* elements. The *SoftwareComposition* type points to the *CompositionType* type which eliminates any nested software components in a *SoftwareComposition* instance. The *SoftwareComposition* type models the architecture of the software components deployed on an ECU, their ports, and the ports' connectors. Software components are modeled using the *ComponentPrototype* type; ports are modeled using the *PPortPrototype* type or *RPortPrototype* type for providing or requiring services; connectors are modeled using the *ConnectorPrototype* type.

The *SystemMapping* type binds the software components to ECUs and the data elements to signals and frames. The *SystemMapping* type aggregates the *SwcToEcuMapping* type, which maps *ComponentPrototype* elements to an *EcuInstance*. According to AUTOSAR, only one *SwcToEcuMapping* instance should be created for every processing unit or memory partition in an ECU.

# 4 GM-to-AUTOSAR Model Transformation

We implement a GM-to-AUTOSAR model transformation to demonstrate the practicality of adopting transformations in the automotive industry. We rationalize our choice of the tool and language and we summarize the pragmatics of the chosen language. We then discuss the transformation rules and implementation details. Our transformation takes as inputs the source GM metamodel, the target AUTOSAR system template, and an input GM model. The output is an AUTOSAR model.

## *4.1* Selecting Model Transformation Tool and Language

Several tools and their accompanying languages have been considered for implementing the transformation including IBM Rational Asset Manager (RAM) [13], the RulesComposer add-on for IBM Rhapsody [14], and MDWorkbench [18].

After investigating the candidate tools, we concluded that IBM RAM and Rules Composer are not suitable for this transformation. RAM is a repository-based tool that offers APIs to create relationships between repository assets (e.g. models). The APIs can manipulate a model as a whole, not the individual model elements. As fine-grained manipulations are essential for our transformation, the support provided by RAM is not sufficient. RulesComposer is a rule-based model-to-text generator. Rules are specified as templates composed of static text and placeholders. When executed, the static text is copied into the output, and the placeholders are extracted from the input models. When defining rules, one must ensure that the template generates well-formed XMI files. Thus, defining the template is time-consuming and error-prone. Moreover, the rule templates can be very verbose, and thus, difficult to maintain.

MDWorkbench is an Eclipse-based tool for developing model-to-model transformations using the Atlas Transformation Language (ATL) [1] or the Model Query Language (MQL) [18]. ATL has declarative and imperative constructs, while MQL has imperative constructs only. MDWorkbench can manipulate models conforming to the metamodels registered in the tool (e.g. AUTOSAR) using rules defined in ATL and MQL. Thus, we choose MDWorkbench to implement the transformation. ATL was chosen rather than MQL because ATL provides flexibility to mix-and-match declarative and imperative constructs in the same rule definition.

## *4.2* ATL Pragmatics

In ATL, a model transformation is defined as a set of rules and helpers. Rules specify the creation of output model elements. Helpers are used to modularize a transformation. ATL defines four types of rules and two types of declarative helpers.

***Rule Types.*** The four types of rules are matched rules, lazy rules, unique lazy rules, and called rules. A matched rule specifies how a source pattern is transformed to a target pattern. Matched rules are executed in the order of their specification and are automatically executed once for each matching pattern. A lazy rule is a rule that is executed only when called for a matching pattern and can be called multiple times for any match in the input model. A unique lazy rule is a rule that is executed only when

called and can be called at most once for any match in the input model. A called rule is a parameterized rule that is executed only when called and creates an element in the output model without matching any source patterns. The four kinds of rules have an optional imperative code block to specify complicated functionality.

Matched rules are suitable for automatic detection of all pattern matches in the input model and creation of their corresponding target patterns; lazy rules and unique lazy rules are suitable for selective pattern matching, with consideration of the number of times these rules should be run; and called rules are suitable for creating output model elements that do not match any input model elements.

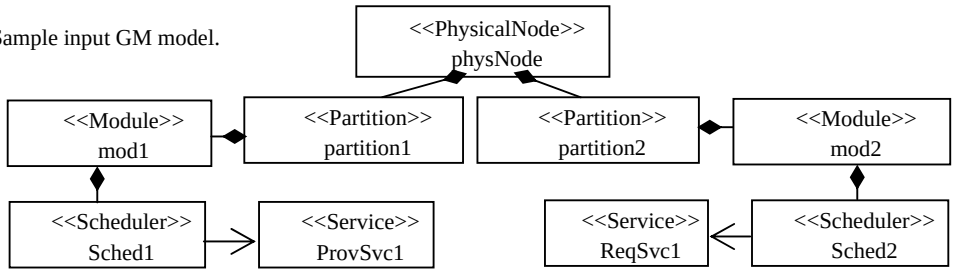
***Helper Types.*** The two types of helpers are functional helpers and attribute helpers. A functional helper is a parametric function that is evaluated each time it is called. An attribute helper is a non-parametric function that is evaluated only in the first call. An attribute helper is more efficient to implement a non-parametric functionality. Otherwise, a functional helper can implement a parametric functionality.

### *4.3* Model Transformation Design and Development

Our transformation rules were crafted in consultation with domain experts at GM to realize the required mappings between the metamodels. For reasons of confidentiality, we present a simplified version of the actual rules. Let *M* be the input GM model and *M'* the to-be-generated output AUTOSAR model. The rules are defined as follows:

1. For every element *physNode* of the *PhysicalNode* type in M, generate an element *sys* of the *System* type, an element *swcompos* of the *SoftwareComposition* type, a containment relation (*sys*, *swcompos*), an element *composType* of the *CompositionType* type, a relation (*swcompos*, *composType*), an element *sysmap* of the *SystemMapping* type, a containment relation (*sys*, *sysmap*) and an element *ecuInst* of the *EcuInstance* type in M';
2. For every element *partition* of the *Partition* type in M, generate an element *swc2ecumap* of the *SwcToEcuMapping* type and a containment relation (*sysmap, swc2ecumap*) in M';
3. For every containment relation (*physNode, partition*) in M, generate a relation (*swc2ecumap, ecuInst*) in M';
4. For every element *mod* of the *Module* type in M, generate an element *comp* of the *ComponentPrototype* type in M';
5. For every containment relation (*partition, mod*) in M, generate a containment relation (*composType, comp*) and a relation (*sw2ecumap, comp*) in M';
6. For every relation (*sched, svc*) of the *provided* type between a *sched* element of the *Scheduler* type and a *svc* element of the *Service* type with a containment relation *(mod, sched)*, generate a *pPort* element of the *PPortPrototype* type and a containment relation (*composType , pPort*) in M';
7. For every relation (*sched, svc*) of the *required* type between a *sched* element of the *Scheduler* type and a *svc* element of the *Service* type with a containment relation (*mod, sched*), generate a *rPort* element of the *RPortPrototype* type and a containment relation (*composType, rPort*) in M'.

(a) Sample input GM model.




(b) Output AUTOSAR model for (a).

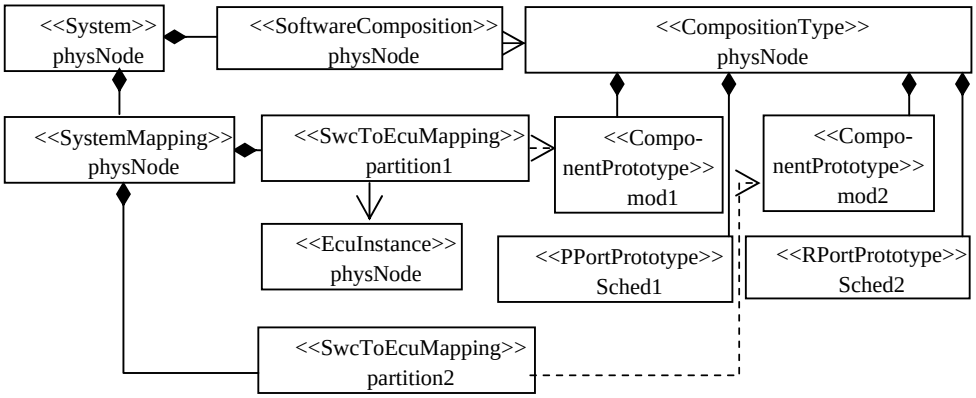



Fig. 4. (a) Sample GM input model and (b) its corresponding AUTOSAR output model.

Fig. 4 demonstrates the required transformation from a sample GM model (Fig. 4 (a)) to its expected output AUTOSAR model (Fig. 4(b)) based on the above mentioned rules. The *PhysicalNode* element is mapped to a *System* element, an *EcuInstance* element, a *SystemMapping* element, a *SoftwareComposition* element, and a *CompositionType* element (Rule 1). The *Partition* elements are mapped to the *SwcToEcuMapping* elements (Rule 2), each of which is associated with the generated *EcuInstance* element (Rule 3). The *Module* elements are mapped to the *ComponentPrototype* elements aggregated by a *CompositionType* element and referred to by their corresponding *SwcToEcuMapping* elements (Rules 4-5). The *Scheduler* element aggregating a provided *Service* is mapped to a *PPortPrototype* element (Rule 6). The other *Scheduler* element is mapped in a similar manner (Rule 7).

The transformation development follows an iterative, incremental process. First, a simple GM model is created in the MDWorkbench model editor. Then, a transformation is implemented to transform the input GM model into an AUTOSAR model. The AUTOSAR model is then validated and if the transformation is correct, the process is repeated with additional types in the input model and additional transformation rules. If the output model contains errors, the transformation is analyzed and fixed.

Validation is performed manually. For an input GM model, an expected output AUTOSAR model is created in the MDWorkbench Model Editor. The transfor-

mation‘s output model is compared with the manually-created model. Equivalence of the models implies a correct transformation.

### *4.4* The Transformation Implementation Using ATL

The GM-to-AUTOSAR transformation contains two ATL matched rules and 9 functional helpers implementing the 7 rules in Section 4.3. We also define 6 attribute helpers to access the model attribute values. Table 1 lists the matched rules and functional helpers and their implemented rules in Section 4.3.

Table 1. Matched rules and functional helpers and the implemented rules.

| Matched Rule (MR)/ Functional Helper (FH) | Corresponding Rules: Section 4.3 |
|---|---|
| MR1: `createComponent` | 4 |
| MR2: `initSysTemplate` | 1 |
| FH1: `initEcuInst` | 1 |
| FH2: `createSwc2EcuMappings`<br>FH3: `initSingleSwc2EcuMapping` | 2-3 |
| FH4: `addComponents` | 5 |
| FH5: `getAllPPortsInEcu`<br>FH6: `createPPort` | 6 |
| FH7: `getAllRPortsInEcu`<br>FH8: `createRPort` | 7 |
| FH9: `getAllSWCinEcu` | 5 |

The matched rule `createComponent` maps *Module* elements to *ComponentPrototype* elements. The matched rule `initSysTemp` maps a *PhysicalNode* element to a *System* element, a *SystemMapping* element, a *SoftwareComposition* element and a *CompositionType* element by calling the 9 functional helpers to implement rules 1-3 and 5-7. The helper `initECUInst` initializes an *EcuInstance* element. The helper `initSingleSwc2EcuMapping` initializes a *SwcToEcuMapping* instance. The helper `createSwc2EcuMappings` creates a list of *Swc2EcuMapping* elements corresponding to all the *Partition* elements in the input model. The helper `getAllSwcInEcu` creates the containment relation between the *CompositionType* elements and the *ComponentPrototype* elements. The helper `addComponents` creates the relation between the *SwcToEcuMapping* elements and their corresponding *ComponentPrototype* elements. The helper `getAllPPortsInEcu` creates a *PPortPrototype* element using the helper `createPPort` for *Schedulers* with at least one provided *Service.* Similar helpers generate *RPortPrototype* elements.

The ATL predefined function `resolveTemp` connects the *ComponentPrototype* elements created by the `createComponent` matched rule to the *CompositionType* elements created by the `initSysTemp` matched rule.

Implementing the transformation revealed some insights on using MDWorkbench and ATL in industrial applications. Both the GM and the AUTOSAR metamodels are complex in structure. To process models conforming to complex metamodels, ATL provides flexibility of using declarative and imperative constructs to implement com-

plex transformations. Moreover, since the output models have many relationships among model elements, decisions on where an element should be created in the transformation such that it will be accessible for the downstream transformation are required. One such example is the relation between the *SoftwareComposition* element and the *ComponentPrototype* element. The transformation can be either specified as one rule or modularized as many rules. Although modularization requires that the order of the rules be consistent with their dependencies, ATL mitigates this drawback through the `resolveTemp` function which allows a rule to reference the elements that are yet to be generated by other rules regardless of their specification order. However, the `resolveTemp` function makes the transformation less readable and difficult to debug, so the function should be used only when necessary.

For validation, sample GM models were created in the MDWorkbench Model Editor, including the model in Fig. 4(a), and were used for evaluation. The output models were verified as described in Section 4.3. The transformation was found to produce the expected output models. Sample GM models were used for validation instead of actual GM models since many of the actual GM models did not conform to the GM metamodel, which represents a major challenge for adopting MDD in industrial environments.

## 5 Discussion

Based on our case study, we present open issues requiring further investigation for successful adoption of model transformations in the automotive industry. Recommendations for MDD tool and language development are also discussed.

### *5.1* Interoperability of MDD tools

One of the major challenges encountered in our study was the lack of interoperability between commercial tools for developing transformations. Specifying the model transformation using ATL was not straightforward due to the formats of the manipulated metamodels. ATL can only manipulate MOF [21] or Ecore [23] metamodels, which the GM metamodel in Rhapsody native format is not compatible with. This required the conversion of the GM metamodel to a compatible format.

MDWorkbench has a Rhapsody connector that allows importing the GM metamodel into MDWorkbench and converting it to Ecore format. To avoid the issue of dual license from different vendors with different licensing policies with such an approach, we addressed the problem using XMI. An Ecore metamodel is essentially an XMI file and Rhapsody has an XMI toolkit to export Rhapsody metamodels to XMI files. Exporting the GM metamodel using the XMI toolkit generated an XMI file that does not conform to the Ecore meta-metamodel. To create an Ecore version, we import the XMI into RulesComposer as a metamodel, which creates an Ecore metamodel and an Eclipse plugin project. Exporting the project from RulesComposer to MDWorkbench as a plugin generates a registered GM Ecore metamodel.

Blanc et al. [5] decomposed the interoperability problem into two concerns: the compatibility of the exchanged models, and the definition of an exchange mechanism. Their study proposed an architecture to address these two concerns. Implementing transformations between tools manipulating models that conform to different metamodels was proposed in [6], [4]. Kolovos et al. [15] proposed a framework that supports composing model management tasks with software development tasks in coherent workflows. Although these solutions have been integrated into IDEs, they are not fully automated in applications. MDD tools and transformation languages deserve further research to support easy integration and interoperability with each other.

### *5.2* Optimization in Model Transformations

Our transformation mapped GM models representing a deployment of the software components on physical nodes to their equivalent AUTOSAR models. The transformation exercised one mapping between the two metamodels and generated an AUTOSAR model reflecting the deployment configuration. From the deployment perspective, there are other design options that may yield a more desirable deployment in the output AUTOSAR model with respect to some utility function.

Solutions exist to support optimization during the transformation. Schätz et al. [22] proposed a formalized approach to explore the design space using rule-based transformations. Intermediate models were represented using a relational formalization and rules were represented using predicates. Drago et al. [9] proposed the QVT-Rational framework to explore design options which optimize quality metrics. First, a domain expert specifies the metamodels to be manipulated, the quality metrics of interest, the quality-prediction tool chain and the method for design feedback generation. Then, a designer specifies desirable values for quality metrics and asks QVT-Rational for design solutions. Tools that target industry use need to support scalable design-space exploration to aid developers in exploring design options of the generated model.

### *5.3* Dealing with Semantic Differences between Metamodels

Identifying which target metamodel elements best represent a given source metamodel element can be a difficult task. Reasons include: (1) the precise semantics of a metamodel may not have been documented sufficiently and only be fully known to metamodel developers themselves; consultation of these developers may be time consuming or even impossible. (2) The lack of support in metamodel evolution often means that the metamodels contain redundancies or inconsistencies. (3) The mapping of source to target elements is dependent on the transformation's purpose, because it determines to what extent aspects of model semantics can be removed (e.g., for abstraction), preserved (e.g., for refactorings) or refined (e.g., for code generation).

To facilitate transformation development, techniques to (1)enforce documenting metamodel semantics, (2) suggest mappings between metamodels using similarity matching or "learning" [17], [20], and (3) validate transformations are of high interest.

## 6 Conclusions and Future Work

In this study, we present a solution to migrating legacy VCS design models using model transformations in the automotive industry. The study has two major goals: (1) exploring the practicality of using model transformations in an industrial context to map between industrial metamodels and (2) benefitting GM by supporting automated convergence to AUTOSAR. The implemented transformation converts domain-specific GM models to their equivalent AUTOSAR models. We discussed the transformation context in the development process. Based on our experiences, we discuss which tool and language are appropriate for implementing the transformation, the challenges encountered and open issues that need further investigation.

Research studies on adopting MDD in industry have been published [19], [23], but a few investigated adopting transformations in industry. Daghsen et al. [8] transformed AUTOSAR timing models to classical scheduling models to perform timing analysis. Giese et al. [12] used triple graph grammars to synchronize between SysML system engineering models and AUTOSAR software engineering models. Our study differs from other studies in that the two manipulated metamodels are complex, industrial metamodels, which allows us to draw realistic conclusions regarding the practicality of adopting transformations in industry. Our study considers the entire transformation development process, from tool and language selection to transformation creation and validation. Future work includes extending the transformation to the full GM metamodel and using white-box or black-box testing [11], [16] for validation.

**Acknowledgements.** This work is supported in part by NSERC, as part of the NECSIS Automotive Partnership with General Motors, IBM Canada and Malina Software Corp.

## 7 References

[1] Atlas Transformation Language – ATL, http://eclipse.org/atl/.
[2] AUTOSAR Consortium. AUTOSAR, http://AUTOSAR.org/.
[3] AUTOSAR Consortium. AUTOSAR System Template, http://AUTOSAR.org/index.php?p=3&up=1&uup=3&uuup=3&uuuup=0&uuuuup=0/AUTOSAR_TPS_SystemTemplate.pdf
[4] Bezivin, J., Brunelière, H., Jouault, F., Kurtev, I. Model engineering support for tool interoperability. In *Workshop in Software Model Engineering(WiSME)*, Montego Bay, Jamaica, 2005.
[5] Blanc, X., Gervais, M.-P., Sriplakich, P. Model Bus: Towards the interoperability of modelling tools. In *Model Driven Architecture: Foundations & Applications (MDAFA)*, Linköping, Sweden, vol. 3599, pp. 17-32, 2004.
[6] Brunelière, H., Cabot, J., Clasen, C., Jouault, F., Bézivin, J. Towards model driven tool interoperability–Bridging Eclipse and Microsoft modeling tools. In *European Conf. on Modeling Foundations & Applications(ECMFA)*, Paris, France, vol.6138, pp.32-47, 2010.

[7] Cottenier, T., Berg, A., Elrad, T. The Motorola WEAVR:Model weaving in a large industrial context. In *Aspect-Oriented Software Development(AOSD)*, Vancouver, Canada, 2007.

[8] Daghsen, A., Chaaban, K., Saudrais, S., Leserf, P. Applying holistic distributed scheduling to AUTOSAR Mmethodology. In *Embedded Real-Time Software & Systems (ERTSS)*, Toulouse, France, 2010.

[9] Drago, M., Ghezzi, C., Mirandola, R. Towards quality driven exploration of model transformation spaces. In *Model Driven Engineering Languages & Systems (MODELS)*, Wellington, New Zealand, pp. 2-16, 2011.

[10] Eclipse Modelling Framework (EMF), http://wiki.eclipse.org/EMF

[11] Fleurey, F., Baudry, B., Muller, P.-A., Le Traon, Y. Qualifying input test data for model transformations. In *Software System Modelling (SoSyM) 8(2)*, pp. 185-203, 2007.

[12] Giese, H., Hildebrandt, S., Neumann, S. Model synchronization at work: Keeping SysML and AUTOSAR models consistent. In *Graph Transformations & Model-Driven Engineering*, vol. 5765, pp.555-579, 2010.

[13] IBM Corporation. IBM Rational Asset Manager (RAM). http://www01.ibm.com/software/rational/products/ram/.

[14] IBM Corporation. IBM Rational Rhapsody. http://www.ibm.com/developerworks/ downloads/r/rhapsodydeveloper/index.html.

[15] Kolovos, D., Paige, R., Polack, F. A framework for composing modular and interoperable model management tasks. In *Model Driven Tool & Process Integration (MDTPI)*, Berlin, Germany, 2008.

[16] Küster, J., Abd-El-Razik, M. Validation of model transformations - First experiences using a white box approach. In *Model Development, Validation & Verification (MoDeVa)*, Genova, Italy, pp.62-77, 2006.

[17] Mandelin, D., Kimelman, D., Yellin, D. A Bayesian approach to diagram matching with application to architectural models. In *Intl. Conf. on Software Engineering (ICSE)*, Shanghai, China, p.222–231, 2006.

[18] Sodius. MDWorkbench, http://www.mdworkbench.com/

[19] Mohagheghi, P., Dehlen, V. Where is the proof? - A review of experiences from applying MDE in industry. In *European Conf. on Model Driven Architecture: Foundations & Applications (ECMDA-FA)*, Berlin, Germany, pp.432-443, 2008.

[20] Nejati, S., Sabetzadeh, M., Chechik, M., Easterbrook, S., Zave, P. Matching and merging of Statechart specifications. In *Intl. Conf. on Software Engineering (ICSE)*, Minneapolis, USA, pp.54-64, 2007.

[21] Object Management Group (OMG): Meta Object Facility (MOF) Specification — Version 1.4, April, 2002.

[22] Schätz, B., Hölzl, F., Lundkvist, T. Design-space exploration through constraint-based Mmodel transformation. In *Engineering of Computer Based Systems (ECBS)*, Oxford, UK, p.173 – 182, 2010.

[23] Steinberg, D., Budinsky, F., Paternostro, M., Merks, E. Chapter 5 Ecore Modeling Concepts. In *Eclipse Modeling Framework 2nd edn*. Addison-Wesley Professional, 2009.

[24] Teppola, S., Parviainen, P., Takalo, J. Challenges in the deployment of model driven development. In *Intl. Conf. on Software Engineering Advances (ICSEA)*, Porto, Portugal, pp.15-20, 2009.