Softw Syst Model
DOI 10.1007/s10270-013-0365-1

SPECIAL SECTION PAPER

Model transformations for migrating legacy deployment models in the automotive industry

Gehan M. K. Selim · Shige Wang · James R. Cordy · Juergen Dingel

Received: 6 November 2012 / Revised: 10 June 2013 / Accepted: 25 June 2013

Abstract Many companies in the automotive industry have adopted model-driven development in their vehicle software development. As a major automotive company, General Motors (GM) 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. Since AUTomotive Open System ARchitecture (AUTOSAR) has been developed as a standard to ease the process of integrating components provided by different suppliers and manufacturers, there has been 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 applying model transformations to address the challenges in migrating GM-specific, legacy models to their AUTOSAR equivalents. As a case study, we have built and validated a model transformation using the MDWorkbench tool, the Atlas Transformation Language, and the Metamodel Coverage Checker tool. 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.

Communicated by Dr. Antonio Vallecillo and Dr. Juha-Pekka Tolvanen.

G. M. K. Selim (✉) · J. R. Cordy · J. Dingel
School of Computing, Queen's University, Kingston,
ON K7L2N8, Canada
e-mail: gehan@cs.queensu.ca

J. R. Cordy
e-mail: cordy@cs.queensu.ca

J. Dingel
e-mail: dingel@cs.queensu.ca

S. Wang
Electrical and Controls Integration Lab, General Motors Research and Development, Warren, MI 48090, USA
e-mail: shige.wang@gm.com

Keywords Model-driven development (MDD) · Model transformations · AUTOSAR · Transformation languages and tools · Automotive control software · Black-box testing

1 Introduction

Model-driven architecture (MDA) [72] is a standardization effort led by the Object Management Group (OMG) for developing systems using platform-independent models. The application of MDA to software systems is referred to as model-driven development (MDD) [12]. MDD is a relatively new software development methodology that uses models as the central means for software specification and communication.

In MDD, the software development process can be conceptually treated as a sequence of model transformations, each of which converts an input model conforming to a source metamodel into an output model conforming to a target metamodel. For example, transformations can be used in MDD to transform abstract models into detailed models (and eventually into deployable code) and to refactor models. Thus, model transformations form a vital part of MDD. Model transformations are implemented using a model transformation language, which can be declarative, imperative, or hybrid. While a declarative language or language construct typically yields a simpler and more compact specification, an imperative language or language construct is more likely to be capable of specifying complex transformations [46].

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 physical nodes on which software is deployed and execution frames. VCS models conforming to this internal, proprietary metamodel have been used in several vehicle production domains at GM, such as body control, access control, and monitoring.

Recently, the AUTomotive Open System ARchitecture (AUTOSAR) [6] has been developed as an industry standard to facilitate integration of software components from different manufacturers and suppliers. AUTOSAR defines its own metamodel with a well-defined layered architecture and interfaces and aims at exchangeability and interoperability among components from different suppliers and manufacturers. Since converging to AUTOSAR is a strategic direction for future modeling activities, transforming GM-specific legacy models to their equivalent AUTOSAR models becomes essential. Model transformation is considered as a key enabling technology to achieve this convergence objective.

Despite the existence of studies in industry adoption of MDD [5,25,55,75], no model transformation is reported to have migrated legacy models in the automotive industry. To increase our understanding and test the practicality of using transformations for migrating legacy models in an industrial environment, we have developed and validated a transformation from a subset of GM-specific legacy models to their equivalent AUTOSAR models with assistance of a commercial model transformation tool and a black-box testing tool.

This paper is an extended version of a study that we previously conducted [70]. In addition to discussing the transformation problem demonstrated in [70], we further validate the transformation in this paper using an existing black-box testing tool, the Metamodel Coverage Checker (MMCC) [45]. We report on the testing results and on the practicality of using such testing tools to validate industrial transformations. In summary, we found no bugs in the transformation, but we discuss a few issues that require attention to facilitate industrial transformation testing such as automating the stages of transformation testing.

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, including its rules specification, implementation, and validation. Section 5 discusses our experiences and issues that require further research. Section 6 summarizes related work. The paper is concluded in Sect. 7 with a summary and future work.

2 VCS development, models, and model transformations

Applying model transformation requires understanding of the development process, which provides a context for the target transformation. For vehicle control software (VCS) development, the relevant process artifacts include design stages and activities, and the input and output models of each stage.

2.1 Typical VCS development process and models

The VCS development process is typically described as a *V*-diagram [63], shown in Fig. 1. In this process, the stages on the left-hand side of the *V*-diagram are activities related to design and implementation, and the stages of the right-hand are activities related to integration and validation. 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 or external organizations for refinement into design models and then implemented by hardware and software components. These

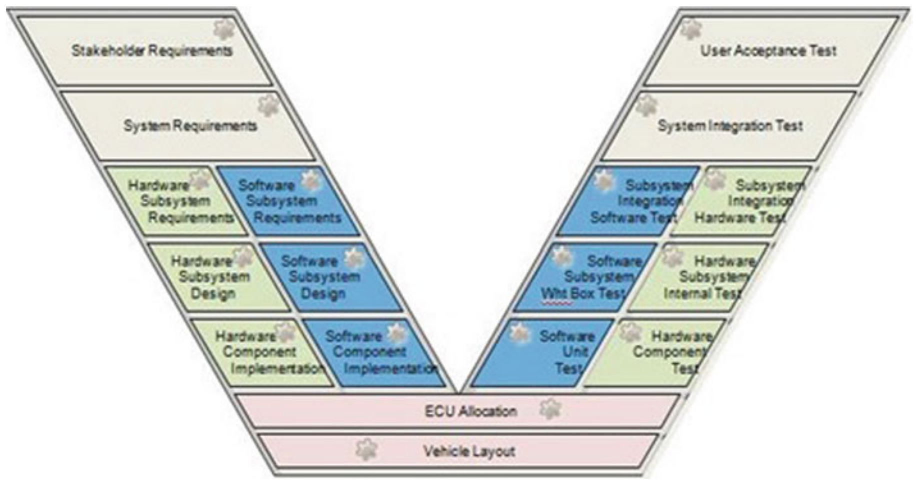

Stakeholder Requirements
System Requirements
Hardware Subsystem Requirements
Software Subsystem Requirements
Hardware Subsystem Design
Software Subsystem Design
Hardware Component Implementation
Software Component Implementation
User Acceptance Test
System Integration Test
Subsystem Integration Software Test
Subsystem Integration Hardware Test
Software Subsystem Wht Box Test
Hardware Subsystem Internal Test
Software Unit Test
Hardware Component Test
ECU Allocation
Vehicle Layout

Fig. 1 *V*-diagram for the VCS development process

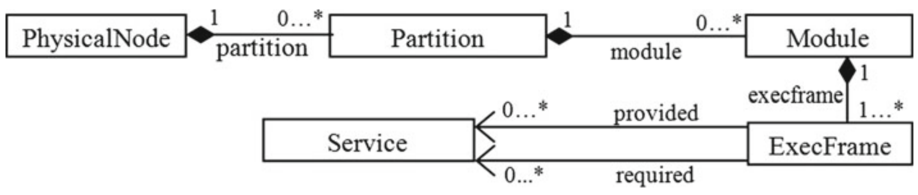


Fig. 2 The subset of the GM metamodel used in our transformation

implemented components are integrated into electronic control units (ECUs), configured for a designated vehicle product. The components are then tested and validated at various levels against their corresponding models on the same level on the left-hand side of the V-diagram.

Different types of models are used and generated in the process, including control models and hardware architecture models. The models use different formalisms: control models use differential equations and timing-variation functions, and hardware architecture models use annotated block diagrams. Selected modeling tools (e.g., Simulink, Rhapsody) and languages (e.g., UML, AADL) are used for modeling.

2.2 Model transformation types in the VCS development process

Given the model types used in the VCS development process, the transformations manipulating these models can be classified into two categories:

- *Horizontal transformations* Horizontal transformations manipulate models at the same abstraction level [54]. Examples include the transformation of a state machine in Matlab Stateflow into a UML 2 state machine. Such transformations are normally used to verify integration when subsystems/components are composed to realize a system function. The modeling languages for the source and target models may have different syntax, but must share similar, or overlap in, semantics.
- *Vertical transformations* Vertical transformations manipulate models at different abstraction levels [54]. Examples include generation of 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.

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 metamodel to GM which we will refer to as the GM metamodel. The target metamodel is the AUTOSAR System Template, version 3.1.5 [7]. To simplify the exercise without losing generality, a subset of the GM metamodel and the AUTOSAR metamodel 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

Figure 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 in a *PhysicalNode* on which software is deployed. Multiple *Module* instances can be deployed on a single *Partition*. The *Module* type defines the atomic deployable, reusable element in a product line and can contain multiple *ExecFrame* instances. The *ExecFrame* type, i.e., an execution frame, models the basic unit for software scheduling. It contains behavior-encapsulating entities and is responsible for managing services provided or required by the behavior-encapsulating entities. Each *ExecFrame* may provide and/or require *Service* instances, which model the services provided or required by the *ExecFrame*.

3.2 The AUTOSAR metamodel

The AUTOSAR metamodel is defined as a set of templates, each of which is a collection of classes, attributes, and relations used to specify an AUTOSAR artifact such as software components and ports. Among the defined templates, the *System* template [7] is used to capture the configuration of a system or an electronic control 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*. Figure 3 shows the metatypes in the ECU Extract that capture software deployment on an ECU. Our transformation manipulates AUTOSAR version 3.1.5.

The ECU extract is modeled using the *System* type that aggregates *SoftwareComposition* and *SystemMapping* elements. The *SoftwareComposition* type points to the

[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.

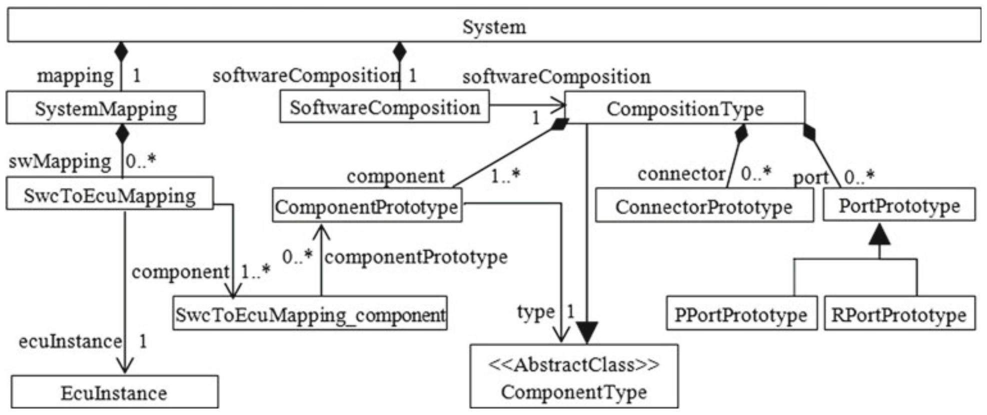


Fig. 3 The AUTOSAR System Template containing relevant types used by our transformation

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, the ports of these software components, and the ports connectors. Each software component is modeled using a *ComponentPrototype*, which defines the structure and attributes of a software component; each port is modeled using a *PortPrototype*, i.e., a *PPortPrototype*or a *RPortPrototype* for providing or requiring data and services; each connector is modeled using a *ConnectorPrototype*. Each *ComponentPrototype* must have a *type* that refers to its container *CompositionType*.

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 assigns *SwcToEcuMapping_component*s to an *EcuInstance*. *SwcToEcuMapping_component*s in turn refer to *ComponentPrototype* elements. According to AUTOSAR, only one *SwcToEcuMapping* should be created for each processing unit or memory partition in an ECU.

4 GM-to-AUTOSAR model transformation

In this case study, we implement a GM-to-AUTOSAR model transformation to demonstrate the practicality of adopting transformations in the automotive industry. First, we rationalize our choice of the model transformation tool and language. Accordingly, we summarize the pragmatics of the chosen language. We then demonstrate the transformation rules needed to map between the two metamodels which were defined in consultation with domain experts. Finally, we discuss the transformation implementation and validation.

Our transformation takes three inputs: the source GM metamodel, the target AUTOSAR system template, and an input GM model. The output of the transformation is an AUTOSAR model.

4.1 Selecting model transformation tool and language

A model transformation language is usually integrated as part of a modeling integrated development environment (IDE). Hence, the choice of a transformation language is tightly coupled with the choice of the supporting IDE. It is desired that the selected tool has strong commercial support. Several tools and their accompanying languages have been considered as candidates for implementing the transformation. The considered tools include IBM Rational Asset Manager (RAM) [41], the RulesComposer add-on for IBM Rhapsody [42], and MDWorkbench [71]. Although several other tools exist, GM was interested in using one of the former three tools due to the availability of their licenses (and thus, their technical support).

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 Java APIs to query assets and to create relationships between repository assets. Assets are a group of artifacts that solve a specific problem. So if a model is stored as an asset, the APIs can manipulate only the model as a whole, not the individual modeling elements. As fine-grained manipulations are essential for mapping between individual modeling elements and implementing our transformation, the transformation support provided by RAM is not sufficient. RulesComposer is a rule-based, model-to-text generator. With RulesComposer, rules are specified as templates composed of static text and placeholders. When executed, the static text is copied into the output file, and the placeholders are extracted from the input models. Since models are essentially XMI files typed by a metamodel, the transformation from GM models to AUTOSAR models can be viewed as a model-to-text transformation from GM models to XMI files conforming to the AUTOSAR metamodel that can be implemented by RulesComposer. When using RulesComposer to implement a model-to-model transformation, the developer needs to specify two aspects in a rule template: the mappings between the source and target metamodels (specified in the placeholders); the static text to be placed in the output XMI file (i.e., XMI headers and the opening and closing tags). Further, the developer needs to ensure that the static text and placeholders in the rule template collectively generate well-formed XMI files. On the other hand, when using a model-to-model transformation engine, the static text (i.e., XMI headers, and the opening and closing tags) are automatically added by the transformation engine and the developer only needs to specify the mappings between the source and target metamodels. Thus, implementing the transformation in RulesComposer (or any other model-to-text transformation engine) 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. MDWorkbench uses the Atlas Transformation Language (ATL) [30,46] or the Model Query Language (MQL) [71] for specifying model transformations. 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. MDWorkbench also provides connectors to different modeling tools. Thus, we choose MDWorkbench to implement the model transformation. To define the transformation rules, ATL was chosen as the transformation language rather than MQL because ATL provides more flexibility to mix-and-match declarative and imperative constructs in the same rule definition.

4.2 ATL pragmatics

The ATL manual [30] and the ATL Zoo [29] are helpful resources when learning ATL. However, since some information is a bit spread out, we found it helpful to summarize ATLs main elements and their use.

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 the source pattern to match in the input model and the corresponding target pattern to create in the output model. Matched rules are automatically executed once for each matching pattern. A lazy rule is executed only when called and can be called multiple times for the same matching pattern. A unique lazy rule is executed only when called and can be called at most once for any matching pattern. A called rule is a parameterized rule that is executed only when called and creates a target pattern without matching any source patterns. All rule types have an optional imperative code block that can be used to specify complicated functionality.

Matched rules are suitable for automatic detection of all matching patterns 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. 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 and is evaluated each time it is invoked. An attribute helper is a nonparametric function and is evaluated only the first time it is invoked. Thus, an attribute helper is more efficient to implement a nonparametric functionality. Otherwise, a functional helper can implement a parametric functionality.

Model transformation specification Similarly to source transformation languages, there are two approaches to specifying transformations in ATL: specifying the transformation as one large rule, or modularizing the transformation using smaller rules and helpers. As in any other transformation language, the two approaches present trade-offs between ease of implementation and efficiency. Building one large rule makes all variables accessible throughout the transformation, so the developer need not worry about the ordering of rules in the transformation specification. However, this approach makes the transformation difficult to maintain and less readable. Modularizing the transformation makes the transformation easier to debug and maintain. However, the developer has to ensure that the rules are specified in an order consistent with the dependencies among rules.

4.3 Model transformation design and development

Our transformation rules were defined to realize the required mappings between the input and output metamodels. The rules were crafted in consultation with domain experts at GM, which was a time-consuming process. For reasons of confidentiality, we present a simplified version of the actual transformation rules defined.

Let M be the input GM model and M' be the to-be-generated output AUTOSAR model. The transformation 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 *swc_comp* of the *SwcToEcuMapping_component* type that refers to an element *comp* of the *ComponentPrototype* type in M';
5. For every containment relation (*partition*, *mod*) in M, generate a containment relation (*composType*, *comp*), a *type* relation (*comp*, *composType*), and a relation (*sw2ecumap*, *comp*) in M';
6. For every relation (*exframe*, *svc*) of the *provided* type between a *exframe* element of the *ExecFrame* type and a *svc* element of the *Service* type with a containment relation (*mod*, *exframe*), generate a *pPort* element of the *PPortPrototype* type and a containment relation (*composType*, *pPort*) in M';
7. For every relation (*exframe*, *svc*) of the *required* type between a *exframe* element of the *ExecFrame* type and a *svc* element of the *Service* type with a containment relation (*mod*, *exframe*), generate a *rPort* element of the *RPortPrototype* type and a containment relation (*composType*, *rPort*) in M'.

We use the example in Fig. 4 to demonstrate the required transformation. Figure 4a shows a sample model from the automotive industry that captures the *BodyControl* controller. *Partition*s running on *BodyControl* include *SituationManagement* and *HumanMachineInterface*. Other possible *Partition*s (not shown) include climate control, vehicle motion control, and human interface. Each *Partition* may have portions on multiple controllers, other than *BodyControl*. *Partition*s may contain multiple *Module*s. For example, *SituationManagement* contains *AdaptiveCruiseControl* and may also contain stop-and-go, parking assistant, blind spot detection, and warning. *HumanMachineInteraction* contains *display* and may also contain chimp control and horn. Each *Module* runs multiple *ExecFrame*s at the same or different rates. *AdaptiveCruiseControl* contains *ComputeDesiredSpeed* and may also contain readACCSet and vehicleSpeedSensing. *Display* contains *DisplaySetSpeed*. *ExecFrame*s invoke *Service*s for variable updates. One *ExecFrame* element, *SetACCDesiredSpeed*, provides a *Service* that is required by the other *ExecFrame* element, *GetACCDesiredSpeed*. The expected output AUTOSAR model based on the above-mentioned rules is shown in Fig. 4b. The *PhysicalNode* element is mapped to an *EcuInstance* element, a *System* 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 has an association with the generated *EcuInstance* element (Rule 3). The *Module* elements are

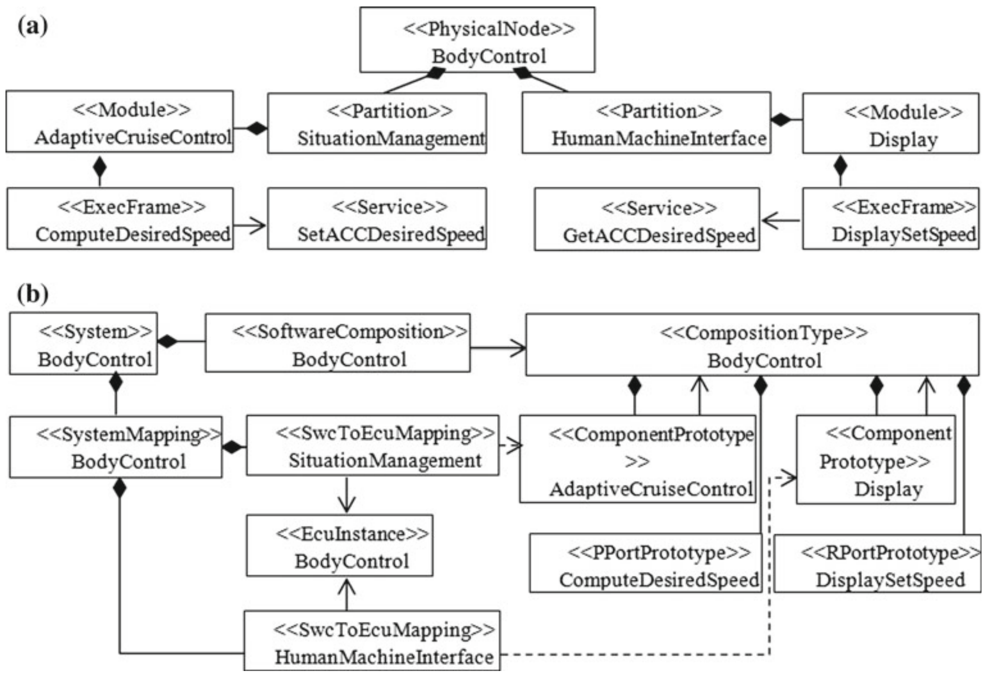

(a)
<<PhysicalNode>> BodyControl
<<Module>> AdaptiveCruiseControl
<<Partition>> SituationManagement
<<Partition>> HumanMachineInterface
<<Module>> Display
<<ExecFrame>> ComputeDesiredSpeed
<<Service>> SetACCDesiredSpeed
<<Service>> GetACCDesiredSpeed
<<ExecFrame>> DisplaySetSpeed
(b)
<<System>> BodyControl
<<SoftwareComposition>> BodyControl
<<CompositionType>> BodyControl
<<SystemMapping>> BodyControl
<<SwcToEcuMapping>> SituationManagement
<<ComponentPrototype>> AdaptiveCruiseControl
<<Component Prototype>> Display
<<EcuInstance>> BodyControl
<<PPortPrototype>> ComputeDesiredSpeed
<<RPortPrototype>> DisplaySetSpeed
<<SwcToEcuMapping>> HumanMachineInterface

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

mapped to the *ComponentPrototype* elements aggregated by a *CompositionType* element (Rules 4–5). The *ComponentPrototype* elements point to their container *CompositionType* element as their *type* (Rule 5). Further, the *ComponentPrototype* elements are referred to by their corresponding *SwcToEcuMapping* elements (Rule 5). The *ExecFrame* element aggregating a provided *Service* is mapped to a *PPortPrototype* element and is aggregated by the *CompositionType* element (Rule 6). The other *ExecFrame* element is mapped similarly (Rule 7).

The development of the model transformation follows an iterative, incremental process. First, a simple GM model is created using the MDWorkbench model editor. Then, a transformation is implemented to transform the input GM model into an equivalent output AUTOSAR model. The output AUTOSAR model is then manually checked to ensure that the transformation performed the required mapping. If the output model is correct, the process is repeated with additional metatypes in the input model and additional rules in the transformation to process these metatypes. If the output model contains errors, the transformation is analyzed, and any erroneous rules or functions are fixed. Validating transformations iteratively after each addition to the transformation's implementation makes locating and fixing bugs easier.

4.4 The model transformation implementation using ATL

The GM-to-AUTOSAR transformation contains two ATL matched rules (Table 1) and 9 functional helpers (Table 2) implementing the 7 rules in Sect. 4.3. We also define 6 attribute helpers to access the model attribute values. We described the rules and helpers in more details in [70]. The relationships between the outputs of the two matched rules are built using the ATL predefined function `resolveTemp` which connects the *ComponentPrototype* elements created by the `createComponent` matched rule to the *CompositionType* element created by the `initSysTemp` matched rule.

4.4.1 Observations on implementing the model transformation using ATL

Implementing the transformation revealed some insights on using MDWorkbench and ATL in industrial applications. Both the GM and the AUTOSAR metamodels are hierarchical and contain many relationships between model elements. To process models conforming to such complex metamodels, ATL provides flexibility of using both declarative and imperative constructs to implement complex transformation rules. Moreover, since the output AUTOSAR models can 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. As discussed in Sect. 4.2, the transformation can be either specified as one rule or modularized as a sequence of rules. Although modularization requires that the order of the rules be consistent with their dependencies, ATL mitigates this downside of modularization through the `resolveTemp` function. The `resolveTemp` function allows a rule to reference the elements that are yet to be generated by other rules at runtime regardless of their order of specification. However, using the `resolveTemp` function makes the transformation less readable and difficult to debug, so the function should be used only when necessary.

4.5 Validation of the model transformation

After implementing the transformation, we used model transformation *testing* [69] to validate the correctness of the transformation. Testing executes a model transformation on input test models or a *test suite* and validates that the generated, actual output model or code matches the expected output model or code [38]. The test suite is built by defining *test adequacy criteria* and building a test suite that achieves coverage of the adequacy criteria [69]. In general, defining test adequacy criteria, and hence testing, can follow a black-box or a white-box approach. Black-box testing assumes that the implementation of the transformation of interest is a *black-box* and builds a test suite based on the specification of the transformation (i.e., source metamodel or contracts). On the other hand, white-box testing assumes that the implementation of the transformation of interest is available and builds a test suite based on the implementation of the transformation.

For our study, we used black-box testing to validate our transformation. More specifically, we used the MMCC tool [45] to facilitate black-box testing. MMCC was imple-

Table 1 Matched rules, their corresponding rules from Sect. 4.3, and their functionality

Matched rule	Corresponding rules: Sect. 4.3	Functionality
`createComponent`	4–5	Maps a *Module* to a *SwcToEcuMapping_component*, and a *ComponentPrototype*
`initSysTemplate`	1	Maps a *PhysicalNode* to a *System*, a *SystemMapping*, a *SoftwareComposition*, and a *CompositionType*

Table 2 Functional helpers, their corresponding rules from Sect. 4.3, and their functionality

Functional helper	Corresponding rules: Sect. 4.3	Functionality
`initEcuInst`	1	Initializes an *EcuInstance* using the name of a *PhysicalNode* as an input
`createSwc2EcuMappings`	2–3	Creates *Swc2EcuMappings* for all *Partition*s in the input model
`initSingleSwc2EcuMapping`	2–3	Initializes a *SwcToEcuMapping* using an *EcuInstance* and a *Partition* as inputs
`addComponents`	5	Creates the relation between a *SwcToEcuMapping* and its *ComponentPrototypes*
`getAllPPortsInEcu`	6	Creates a *PPortPrototype* for any *ExecFrame* that has at least one provided *Service*
`createPPort`	6	Initializes one *PPortPrototype*
`getAllRPortsInEcu`	7	Creates a *RPortPrototype* for any *ExecFrame* that has at least one required *Service*
`createRPort`	7	Initializes one *RPortPrototype*
`getAllSWCinEcu`	5	Creates the containment relation between *CompositionType*s and *ComponentPrototype*s

mented in Kermeta [44] as part of a study by Fleurey et al. [32]. MMCC guides the user in building a test suite based on a predefined test adequacy criterion and a source metamodel.

We used testing over formal validation techniques (e.g., theorem proving or model checking) for several reasons. First, unlike formal validation techniques that use a formalization such as Maude, testing does not require that the user has a thorough knowledge of the formalization. Second, testing has the advantage of uncovering bugs while maintaining a low computational complexity [37]. Finally, formal validation of ATL transformations is a topic of ongoing research. Several studies addressed such problems by reimplementing their transformations in other formalizations such as Petri Nets [62] and Maude [23] to use their accompanying validation techniques [17,49,64,76]. Reimplementing industrial-size transformations in a different formalization can be infeasible due to time and money constraints. Thus, we had to use a validation technique and tool that can process ATL transformations. MMCC is one of the few publically available tools that can validate transformations in any formalization (including ATL) since it is a black-box testing tool (i.e., transformation language independent). We are currently conducting another case study to evaluate our GM-to-AUTOSAR transformation using a prototype developed by Büttner et al. [21] for validating specifically ATL transformations.

4.5.1 Metamodel Coverage Checker (MMCC)

Metamodel Coverage Checker runs on two phases. In the first phase, the user specifies the source metamodel and an adequacy criterion as inputs. In this phase, MMCC uses category-partitioning [61] to *partition* the values of multiplicities and attributes of type integer, string, or boolean into ranges as follows:

– Integer attribute values and multiplicity values are partitioned into three ranges: {0}, {1}, and {>1}.
– String attribute values are partitioned into two ranges: {“”} and {“+”} (i.e., an empty string and a non-empty string).
– Boolean attribute values are partitioned into two ranges: {true} and {false}.

We updated MMCC to generate partitions for attributes that are of types other than integer, string, or boolean. For example, float attributes were partitioned into three ranges: {0}, {(0,1]}, and {>1}.

Using the generated partitions and the specified adequacy criterion, MMCC generates *object fragments* and *model fragments*. An object fragment is a template for a class object that specifies constraints on the values of the attributes and multiplicities of objects from the corresponding class. A model fragment is a template for an input test model that contains one or more object fragments. A model fragment is satisfied by a test model if the objects in the test model satisfy the object fragments in the model fragment.

In the second phase, the user specifies the location of a test suite and MMCC evaluates the test suite by identifying how many model fragments generated in the first phase were satisfied by the test suite. MMCC further generates a summary of the missing model fragments in the test suite to guide the user in building additional test models.

4.5.2 Validation results

For the first phase, we specified two inputs to run MMCC: the GM metamodel and the *AllPartitions* criterion. The *AllPartitions* criterion is a criterion implemented in MMCC and mandates that values from all ranges of each property or multiplicity partition should be represented simultaneously in the same model fragment. For example, for an integer attribute, one model fragment mandates that the attribute should have values from the three integer ranges ({0}, {1}, and {>1}) in a single input model. In this phase, MMCC generated 196 partitions for 196 different attributes and multiplicities values. Accordingly, 196 model fragments were generated for the *AllPartitions* criterion.

Besides the *AllPartitions* criterion, the *AllRanges* criterion was also implemented in MMCC. The *AllRanges* criterion mandates that values from each range of each property or multiplicity partition should be represented in a model fragment. We used the *AllPartitions* criterion instead of the *AllRanges* criterion since it subsumes the *AllRanges* criterion, i.e., a test suite that satisfies the *AllPartitions* criterion also satisfies the *AllRanges* criterion, but the inverse is not true.

We did not run the second phase of MMCC since we started off with an empty test suite. Thus, we need to build a test suite with models that satisfy the 196 model fragments. Having 196 model fragments implies that the test suite can contain at most 196 models to satisfy the *AllPartitions* criterion. However, one model can cover more than one model fragment at a time. Thus, we manually built a test suite of 100 test models to cover the 196 model fragments.

Our model transformation was executed using the generated test suite. For each test model in the test suite, the corresponding output model was verified by manually checking whether the output AUTOSAR model is a valid equivalent of the input GM model. The transformation was found to produce the expected output models for the 100 input test models.

Actual GM models were not used for validation since many of the actual GM models did not conform to the GM metamodel. They were built using IBM Rational Rhapsody [42] which allows building models without mandating that these models be valid instances of a specific metamodel (i.e., Rhapsody does not check conformance of the GM models to the GM metamodel). Since migrating to AUTOSAR is unavoidable for GM, this migration can be done in two ways. The first alternative is to manually build the AUTOSAR equivalents of all the models to be migrated. The major drawback of this alternative is that different engineers may have different understandings of AUTOSAR and the migration may be inconsistent for different models. The second alternative is to update the GM models to ensure that they conform to the GM metamodel and then using our transformation to migrate all GM models (conforming to the GM metamodel) to their AUTOSAR equivalents in an automated, consistent way. The second alternative is easier to adopt since changing the GM models to conform to the GM metamodel can, in many cases, involve minor changes (e.g., updating an association, adding an attribute name) which is much simpler than building AUTOSAR models from scratch (as in the first alternative) and ensuring that they convey the intended meaning. Thus, to safely adopt the second alternative, we conducted several meetings with GM domain experts to ensure that we implemented the correct mapping between the two metamodels and we validated the implemented transformation using black-box testing.

5 Discussion

Based on our experiences with the GM-to-AUTOSAR transformation, we present some open issues requiring further investigation for successful adoption of model transformations in the automotive industry. Recommendations for future MDD tool and language development are also discussed.

5.1 Interoperability of MDD tools

5.1.1 Observations

One of the major challenges encountered in our study was the lack of interoperability between commercial tools in supporting implementation of model transformations. According to our evaluation of the languages for model transformation implementation, ATL seemed to be an appropriate choice. However, specifying the model transformation rules between the two metamodels using ATL was not straightforward due to the formats of these metamodels. ATL can only be used to create model transformations that manipulate MOF [60] or Ecore [31,73] metamodels, which the GM metamodel in Rhapsody native format is not compatible with. Such incompatibility between the format required by ATL and the format of the GM metamodel required the conversion of the GM metamodel to a compatible format.

To do so, the MDWorkbench tool suite provides a Rhapsody connector that allows importing the GM metamodel into MDWorkbench and converting it to Ecore format. To avoid dual licensing from two different vendors[2] with such an approach, we addressed the problem using XMI. An Ecore metamodel is essentially an XMI file. Rhapsody provides an XMI toolkit to export Rhapsody models and metamodels to XMI files. Thus, we exported the GM Rhapsody metamodel using the XMI toolkit. However, the generated XMI file does not conform to the Ecore meta-metamodel. To create an Ecore version accessible to MDWorkbench, we import the XMI into RulesComposer as a metamodel, which creates an Ecore version of the metamodel and an Eclipse plugin project. Exporting the project from RulesComposer to

[2] The Rhapsody connector provided by MDWorkbench requires a *combined* license from both Sodius (providers of MDWorkbench) and IBM (providers of Rhapsody).

MDWorkbench as a plugin generates a registered GM Ecore metamodel.

5.1.2 Other proposed solutions in the literature

In addition to our solution, there are other solutions to the interoperability problem. Blanc et al. [16] decomposed the interoperability problem into two concerns: ensuring the compatibility of the model formats exchanged between different tools, and defining an exchange mechanism that can be realized at run-time. Their study proposed the *Model Bus* architecture to address these two concerns. In terms of the two interoperability concerns identified in their study, the interoperability problems that we encountered and proposed solutions for are related to the compatibility of the exchanged model formats. Bruneliére et al. [20] and Beźivin et al. [13] proposed implementing model transformations or *bridges* between tools manipulating models that conform to different metamodels. Kolovos et al. [48] proposed a framework that supports composing model management tasks with software development tasks in coherent workflows. Other studies defined frameworks, standards, or guidelines to facilitate interoperability between different tools. For example, projects such as the iFEST project [43] and the CESAR project [22] proposed different frameworks and standards that can be adopted by the industry for the development of embedded systems. To the same end, Broy et al. [19] discussed the *ingredients* required to achieve seamless integration between isolated tools. The study also justified why such a solution has not been implemented so far and the steps required to get closer to building an environment that allows easy integration of different tools and languages.

5.1.3 Future requirements

While a few solutions to the interoperability problem have been implemented in some IDEs, they are not fully automated in practical applications. This can be addressed by using the work done by the iFEST project [43], the CESAR project [22], and Broy et al. [19] as a guide for realizing frameworks that support automated integration between different tools.

5.2 Optimization in model transformations

5.2.1 Observations

Our implemented transformation mapped GM models representing a deployment of the software components on physical nodes to their equivalent AUTOSAR models. The transformation exercised one rigid mapping between elements of the two metamodels and generated an AUTOSAR output model reflecting the deployment configuration. From the deployment perspective, there are other design options we have not explored that may yield a more desirable deployment in the output AUTOSAR model with respect to some utility function.

5.2.2 Other proposed solutions in the literature

Solutions exist to support optimization during the model transformation. For example, Schätz et al. [67] proposed a formalized approach to explore the design space using rule-based model transformations. The study argued that system development is a series of constrained design steps that successively refine a model. Instances of intermediate models were represented using a relational formalization, and transformation rules were represented using predicates. The approach was applied in an automotive-industrial context for implementing a transformation that maps components to units and communication channels to buses. The study argued that the efficiency of the design space exploration approach can be improved. Drago et al. [28] proposed the QVT-Rational framework that explores different design options which optimize predefined 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 different design solutions. The study concluded that QVT-Rational is inefficient for interactive development and cannot guarantee generating an optimal solution for large design spaces.

5.2.3 Future requirements

Future model transformation tools that target industry use need to support scalable design space exploration to aid developers in exploring design options that optimize functional or non-functional requirements of the generated model.

5.3 Dealing with semantic differences between metamodels

5.3.1 Observations

Identifying which elements of the target metamodel best represent a given element in the source metamodel can be a very difficult task. Reasons include the following: (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 and error prone or even impossible. (2) The lack of analysis support in metamodel construction and evolution often means that the metamodels contain redundancies or inconsistencies. (3) The mapping of source to target elements is very dependent on the context and purpose of the transformation, because they determine to what extent aspects of the semantics of model elements can be removed (to, e.g., facilitate a model analysis), or need to be preserved (e.g., for model refactorings) or refined (e.g., for code generation).

5.3.2 Other proposed solutions in the literature

Several studies discussed dealing with semantic differences between metamodels by supporting mapping between the metamodels. Kent and Smith [47] proposed a set of requirements needed in mapping functions (e.g., supporting bidirectional mappings) and tools (e.g., supporting consistency checking of mappings) that are intended to map between different metamodels. Hausmann [39] proposed extending a metamodeling language with additional declarative constructs to express mappings between metamodels. Thus, the language can be used to build metamodels and formally define metamodel mappings while abstracting from transformation direction and platform-specific implementations. Claypool et al. [24] proposed an architecture for a model management system in which mappings between metamodels are captured as models to facilitate tooling and automated mapping between instance models. Maskeri et al. [52] proposed abstracting or *stamping-out* metamodels into their composite software patterns (e.g., multiple inheritance patterns or allowed reference patterns) and defining mappings between the patterns. Thus, the mappings can be reused later on between models conforming to the stamped-out metamodels.

5.3.3 Future requirements

The reviewed studies focused on developing metamodeling techniques and tools that allow defining mappings when defining the metamodels. Little attention has been given to investigating techniques and tools that support mapping between existent metamodels. To facilitate transformation development, techniques to (1) document the semantics of elements during metamodel construction, (2) find and suggest mappings between metamodels using similarity matching or *learning* [51,58], and (3) validate transformations via testing and analysis are of high interest.

5.4 Validating model transformations

5.4.1 Observations

As explained in Sect. 4.5, we used a black-box testing tool to facilitate the validation of our model transformation. After manually examining the model fragments generated by MMCC and the corresponding test models built to satisfy the model fragments, we found that only 45 model fragments out of the 196 actually trigger any rule in our model transformation. The generation of redundant model fragments and the possibility of the test suite not triggering all the rules in the transformation are due to the nature of black-box testing in general; test cases are generated independent of the model transformation implementation.

5.4.2 Other proposed solutions in the literature

More rigorous validation techniques and tools are desirable, especially for safety-critical systems. Formal verification techniques are an active research topic [68] and are not entirely mature yet as discussed in Sect. 4.5. These techniques use formalizations (e.g., graph rewriting systems [65], Petri Nets [62], Maude[23]) to represent transformations and analyze them using analysis specific to those formalizations, e.g., [2,11,17,49,57,64,76]. However, formal verification techniques tend to be computationally expensive and not necessarily scalable [37] and techniques and tools are needed that can handle industrial-size transformations and input models with reasonable resources and time. Further, many of the formal verification techniques developed so far mandate that the user has a strong mathematical background and hence are not easy to use by all developers or testers. Rivera et al. [64] addressed this issue by using graph rewriting systems [65] as a front-end to a tool that analyzes transformations using Maude. Maude [23] is a language that supports Membership Equational Logic (MEL) [18] and has strong support for analysis techniques. The study used graph rewriting systems as a front-end since a graphical representation of a transformation is more intuitive than a textual one. Thus, the study takes advantage of the Maude analysis techniques while making the tool easy to use by developers and testers.

5.4.3 Future requirements

Metamodel Coverage Checker helped provide a systematic way to generate a test suite, but the actual generation and execution of the test models was performed manually and hence was time consuming and error prone. For testing to scale up to industrial-size transformations and models, it is desirable to increase the level of automation in generating the test suite, executing the transformation of interest using the test suite, and evaluating the results of executing the model transformation (e.g., using model differencing).

5.5 Model transformation scalability

5.5.1 Observations

As discussed in Sect. 4.5.2, actual GM models were not used for validation due to their non-conformance to the GM metamodel. Thus, no scalability study was conducted to ensure that the proposed approach scales when used to migrate actual GM models are naturally bigger and more complicated than the models we used to test our transformation (discussed in Sect. 4.5.2).

5.5.2 *Other proposed solutions in the literature*

A few studies reasoned about the scalability of ATL transformations. As discussed in Sect. 6.1, Biehl and Törngren [15] conducted a case study on an automotive brake-by-wire system to demonstrate how transformations in ATL and Tiger EMF can be used to model design decisions. The approach was found to be feasible although no detailed results were demonstrated. Aziz [8] conducted an *exploratory case study* at Ericsson to investigate three model transformation technologies (ATL, IBM TF, and Acceleo). The case study involved interacting with personnel to verify the quality of the three transformation technologies. ATL was found to be the most scalable transformation technology. Syriani [74] claimed that hybrid model transformation paradigms, such as ATL, scale better than graph transformation languages.

5.5.3 *Future requirements*

Based on the above-mentioned studies, we expect that extending our ATL transformation to cover the full scope of the GM metamodel will scale when exercised on actual GM models. Nevertheless, we still need to conduct a scalability study once the GM models are updated to conform to the GM metamodel (as described in Sect. 4.5.2). Further, more empirical studies are needed to quantitatively compare different transformation paradigms in terms of different properties (e.g., scalability) [14]. While Gardner et al. [35] gave an initial comparison of the scalability of different transformation paradigms, not all transformation paradigms were covered (e.g., graph transformations) and no case studies were conducted to provide a quantitative evaluation of properties such as scalability.

6 Related work

We survey studies that investigate using transformations in industry for different purposes and studies that validate transformations used in an industrial context. Then, we discuss how our study is different from the surveyed ones.

6.1 Model transformations in industry

Research studies on adopting MDD in industry have been published [5,25,55,75], but only a few investigated the adoption of model transformations in the industry for different purposes.

Transformations have been used in industry to facilitate analysis. For example, Daghsen et al. [26] transform AUTOSAR timing models into classical scheduling models on which timing analysis was performed. The approach was applied to an industrial steering-by-wire system. Focusing only on timing analysis, the transformation reported did not include details of the tools and languages used to implement the transformation, the challenges encountered, the target metamodel, or how the mapping between the source and target metamodels was obtained. Similarly, Anssi et al. [4] transformed an AUTOSAR scheduling analysis model into its corresponding MAST model to perform timing analysis using the MAST scheduling tool. The approach was applied to a cruise control system. Focusing only on timing analysis, the study only described the mapping required between AUTOSAR scheduling analysis models and MAST models, but no further details were given about the development of the transformation.

Transformations have also been used in industry for model management tasks, e.g., consistency checking between related models. Giese et al. [36] proposed using triple graph grammars to maintain consistency and synchronization between system engineering models in SysML and software engineering models in AUTOSAR. Salay et al. [66] used macromodels for managing related models in the automotive industry. Macromodels represent related models with their relationships captured as formal mappings and constraints. A case study was conducted on the flow diagrams representing an industrial, vehicle product line subsystem. The authors demonstrated how the used macromodels helped uncover inconsistencies and incompleteness of some of the defined models.

Similar to our study, transformations have also been used in other studies for migration in an industrial context. Fleurey et al. [33] demonstrated how the Sodifrance company has been using several model-driven engineering tasks (i.e., automated analysis of code, reverse engineering, model transformations, and code generation) for migration projects using a tool developed in-house called Model-In-Action (MIA). A case study was conducted and MIA was used to migrate a large-scale banking system. Since migration was performed by a competitive company, no details were provided on the development of the transformations used for migration (i.e., specific tools and languages used to build the transformations, the mapping rules that must be realized by the transformation, and transformation implementation details). Similarly, Doyle et al. [27] used model transformations to migrate a sample of legacy domain-specific models (DSMs) to UML models (or other MOF-compliant models) at a financial services company, Fortis. Since an MOF-based source metamodel of the DSMs was not available, a considerable part of the study was dedicated to demonstrating how such a metamodel is derived and how new models conforming to the derived metamodel can be built automatically to reflect the information that was present in the original DSMs. Thus, no details about the development of the migration transformation were provided.

Model transformations have also been used in an industrial context for several other purposes. Biehl and Törngren [15] modeled design decisions using transformations to overcome the knowledge vaporization problem, i.e., the loss of knowledge inherent in design decisions. A case study was conducted on an automotive brake-by-wire system using ATL and Tiger EMF for representing design decisions. Based on the case study, the approach was found to be feasible although no detailed results were given. Ali et al. [1] used a series of transformations to automate test case generation in model-based testing. Two industrial case studies were conducted on a multi-media conferencing system and a safety monitoring component in a safety-critical control system. Hemel and Kats [40] used model transformations for code generation, i.e., input models are transformed into a model of the target program to enable seamless extension of the target language with additional features by extending the output models of the target program. A case study was conducted where the approach was used to implement WebDSL, a domain-specific language for building web applications. The approach was found to have several advantages, e.g., ensuring syntactical correctness of the output model representing the target program and facilitating further transformations on the output model representing the target program.

6.2 Validating model transformations in industry

In general, black-box testing can be based on metamodel coverage or contract coverage [69], i.e., the test suite generated to test the transformation of interest aims to achieve coverage of the source metamodel elements or the transformation contracts. For black-box testing based on metamodel coverage, several studies proposed test adequacy criteria for different metamodels such as the metamodels of class diagrams [3,32,34] and state charts [59,77]. However, very few studies have evaluated the efficiency of the different adequacy criteria [53]. Fewer studies investigated black-box testing based on contract coverage [9,10]. Selim et al. reviewed the state of the art in model transformation testing [69] and other techniques for analyzing and validating model transformations such as formal verification techniques [68].

From the case studies discussed in Sect. 6.2, only a few investigated validating the correctness of their transformation. Fleurey et al. [33] briefly mentioned that the migration was evaluated using a non-regression testing process in which customers were responsible for providing the test cases. Although their tool (i.e., MIA) automates several steps, migration of their subject banking system took 3,500 days of work (i.e., 45 % of the project's cost) since a non-trivial fragment of the migration was performed manually. Details about the testing carried out were not discussed (i.e., criteria used for test case generation, the number of generated test cases, and the results of testing). Giese et al. [36] manually invoked OCL constraint checks to validate that the models manipulated by their transformation preserved well-formedness constraints.

Some studies evaluated their work in terms of other measures, e.g., execution time and size of the output models. Doyle et al. [27] evaluated their migration transformation in terms of the size of the generated models and the execution time of the migration. Ali et al. [1] assessed their transformation in terms of the execution time only. The study concluded that improving the efficiency of the approach is necessary since executing the transformations took almost 6 h for their second case study.

The rest of the transformations were not evaluated at all. The case studies conducted by Daghsen et al. [26] and Anssi et al. [4] did not discuss validation of their transformation; they only focused on validating the schedulability of the transformation's output. The macro models developed by Salay et al. [66] were used for model management but the study did not verify that the built macromodels capture the intended relations between the different models. Biehl and Törngren [15] and Hemel and Kats [40] discussed the advantages and feasibility of their approaches without discussing the validation of their subject transformations.

The difference between our study and surveyed studies Our study differs from other studies reported in the literature in three major aspects. First, to the best of our knowledge, no other case study was conducted to migrate legacy models in the automotive industry which is a significant gap in the literature given the conversion of many original equipment manufacturers (OEMs) to using AUTOSAR. Our transformation migrates GM legacy models to their equivalent AUTOSAR models. The two metamodels manipulated in our study are industrial metamodels, which allow us to draw more realistic conclusions with regard to the practicality of adopting transformations in industry.

Second, although other studies investigated using transformations for migration in other industries (e.g., [27,33]), our study considered in detail the entire transformation development process, from tool and language selection to transformation creation and validation.

Third, to the best of our knowledge, no other industrial case study discussed using testing for validating transformations. Fleurey et al. [33] briefly mentioned that non-regression testing was used to validate their migration transformation but the study did not discuss details of the testing process (i.e., criteria used for test case generation, the number of generated test cases, and the results of testing). We used a black-box testing tool (i.e., MMCC) to validate our transformation, we discussed the test case generation criterion used and we reported on the testing results.

7 Conclusion 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 and easy 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, the selection of the ATL transformation language and the MDWorkbench tool for the transformation implementation, and the development and validation of the model transformation. Based on our experiences, we discuss which tools and languages are appropriate for implementing and validating the transformation, the challenges encountered in the study, and open issues that need further investigation. Thus, our study can be used by other automotive OEMs to guide them in migrating legacy models using model transformations by demonstrating the approach to follow, the suitable MDD tools and languages to use and possible issues that will be encountered and their solutions. Further, the approach we used to map between subsets of the two metamodels can be reused to extend the transformation to the full scope of the metamodels. In other words, we will need to consult domain experts to craft the transformation rules required to map between the full scope of the two metamodels (as described in Sect. 4.3), implement those rules following our iterative and incremental development approach (as described in Sects. 4.3 and 4.4), and test the transformation using the MMCC tool (as described in Sect. 4.5).

Since we demonstrated the effectiveness of our approach for migrating a subset of the GM metamodel to its AUTOSAR equivalent, engineers at GM expressed their interest in extending the transformation to the full scope of the GM metamodel. Thus, future work includes extending the transformation and updating the actual GM models to conform to the GM metamodel, so that the transformation can be used in practice for migrating GM models. While our current transformation covers a substantial subset of the two metamodels that represent the deployment problem, several obstacles might be introduced in such an extension:

1. We expect that significant time and effort will be spent in understanding the remainder of the two metamodels from domain experts.
2. The transformation's complexity will probably increase as more mappings are added. Thus, updating the transformation to make it comprehensible might be necessary (e.g., using more declarative constructs that are more intuitive and readable)
3. As more mappings are added, the transformation may need to be refactored to make generated output elements easily accessible to the added rules and helpers.

Work on validating the transformation can be extended in two directions. First, several steps in the testing process can be automated, e.g., the generation of a test suite, using mutation analysis [56] for test suite assessment, the execution of the model transformation on the test suite, and the evaluation of the output models generated by the model transformation. White-box testing (e.g., [50]) can also be used for validation. Second, formal model transformation verification techniques [68] can be investigated for verifying the transformation.

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

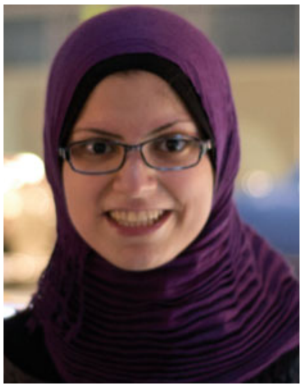
References

1. Ali, S., Hemmati, H., Holt, N., Arisholm, E., Briand, L.: Model transformations as a strategy to automate model-based testing-A tool and industrial case studies. In: Simula Research Laboratory, Technical Report (2010–01). Citeseer (2010)
2. Anastasakis, K., Bordbar, B., Küster, J.: Analysis of model transformations via alloy. In: Model-Driven Engineering, Verification and Validation (MoDeVVa), pp. 47–56 (2007)
3. Andrews, A., France, R., Ghosh, S., Craig, G.: Test adequacy criteria for UML design models. Softw. Test. Verif. Reliab. **13**, 95–127 (2003)
4. Anssi, S., Tucci-Piergiovanni, S., Kuntz, S., Gérard, S., Terrier, F.: Enabling scheduling analysis for AUTOSAR systems. In: International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC), pp. 152–159. IEEE (2011)
5. Aranda, J., Damian, D., Borici, A.: Transition to model-driven engineering: what is revolutionary, what remains the same. In: Model Driven Engineering Languages and Systems (MODELS), pp. 692–708. Springer, Berlin (2012)
6. AUTOSAR: AUTOSAR, http://AUTOSAR.org/ (2007)
7. AUTOSAR.: AUTOSAR System Template, http://AUTOSAR.org/index.php?p=3&up=1&uup=3&uuup=3&uuuup=0&uuuuup=0/AUTOSAR_TPS_SystemTemplate.pdf (2007)
8. Aziz, K.: Evaluating Model Transformation Technologies: An Exploratory Case Study. Department of Computer Science and Engineering, University of Gothenburg, Gotheburg (2011)
9. Bauer, E., Küster, J.: Combining specification-based and code-based coverage for model transformation chains. In: Theory and Practice of Model Transformations, pp. 78–92 (2011)
10. Bauer, E., Küster, J., Engels, G.: Test suite quality for model transformation chains. In: Objects, Models, Components, Patterns, pp. 3–19. Springer, Berlin (2011)
11. Becker, B., Lambers, L., Dyck, J., Birth, S., Giese, H.: Iterative development of consistency-preserving rule-based refactorings.

In: International Conference on Theory and Practice of Model Transformations (ICMT), pp. 123–137 (2011)
12. Beydeda, S., Book, M., Gruhn, V.: Model-Driven Software Development, vol. 15. Springer, Berlin (2005)
13. Bézivin, J., Bruneliere, H., Jouault, F., Kurtev, I.: Model engineering support for tool interoperability. In: Workshop in Software Model Engineering (WiSME), vol. 2. Montego Bay, Jamaica (2005)
14. Bézivin, J., Dupé, G., Jouault, F., Pitette, G., Rougui, J.E.: First experiments with the ATL model transformation language: transforming XSLT into XQuery. In: Workshop on Generative Techniques in the context of Model Driven Architecture (2003)
15. Biehl, M., Törngren, M.: An executable design decision representation using model transformations. In: Software Engineering and Advanced Applications (SEAA), pp. 131–134. IEEE (2010)
16. Blanc, X., Gervais, M., Sriplakich, P.: Model bus: towards the interoperability of modelling tools. In: Model Driven Architecture: Foundations and Applications (MDAFA), pp. 17–32 (2005)
17. Boronat, A., Heckel, R., Meseguer, J.: Rewriting logic semantics and verification of model transformations. In: 12th International Conference on Fundamental Approaches to Software Engineering (FASE), York, UK, pp. 18–33. Springer, Berlin (2009)
18. Bouhoula, A., Jouannaud, J., Meseguer, J.: Specification and proof in membership equational logic. In: Theoretical Computer Science: Trees in Algebra and Programming, vol. 236, pp. 35–132. Elsevier, Amsterdam (2000)
19. Broy, M., Feilkas, M., Herrmannsdoerfer, M., Merenda, S., Ratiu, D.: Seamless model-based development: from isolated tools to integrated model engineering environments. In: Proceedings of the IEEE, vol. 98, pp. 526–545. IEEE (2010)
20. Brunelière, H., Cabot, J., Clasen, C., Jouault, F., Bézivin, J.: Towards model driven tool interoperability: bridging eclipse and microsoft modeling tools. In: European Conference on Modelling Foundations and Applications (ECMFA), vol. 6138, pp. 32–47. Paris, France (2010)
21. Büttner, F., Egea, M., Cabot, J., Gogolla, M.: Verification of ATL transformations using transformation models and model finders. In: International Conference on Formal Engineering Methods (ICFEM) (2012)
22. CESAR. http://www.cesarproject.eu/index.php?id=9
23. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martı-Oliet, N., Meseguer, J., Talcott, C.: All about Maude-A high-performance logical framework, how to specify, program and verify systems in rewriting logic. volume 4350 of LNCS, vol. 4, pp. 50-88. Springer, Berlin (2007)
24. Claypool, K.T., Rundensteiner, E.A., Zhang, X., Hong, S., Kuno, H., Lee, W.-c., Mitchell, G.: Sangam-A solution to support multiple data models, their mappings and maintenance. In: ACM SIGMOD International Conference on Management of Data, vol. 30, p. 606. ACM, New York (2001)
25. Cottenier, T., Van Den Berg, A., Elrad, T.: The motorola WEAVR: model weaving in a large industrial context. In: Aspect-Oriented Software Development (AOSD), vol. 32. Vancouver, Canada (2007)
26. Daghsen, A., Chaaban, K., Saudrais, S., Leserf, P.: Applying holistic distributed scheduling to AUTOSAR methodology. In: Embedded Real-Time Software and Systems (ERTSS). Toulouse, France (2010)
27. Doyle, D., Geers, H., Graaf, B., Van Deursen, A.: Migrating a domain-specific modeling infrastructure to MDA technology. In: International Workshop on Metamodels, Schemas, Grammars, and Ontologies for Reverse Engineering (ateM), Genoa, Italy (2006)
28. Drago, M., Ghezzi, C., Mirandola, R.: Towards quality driven exploration of model transformation spaces. In: Model Driven Engineering Languages and Systems (MODELS), pp. 2–16. Wellington, New Zealand (2011)
29. Eclipse. ATL Zoo, http://www.eclipse.org/m2m/atl/atltransformations/ (2012)
30. Eclipse. Atlas Transformation Language—ATL, http://eclipse.org/atl/ (2012)
31. Eclipse. Eclipse Modelling Framework (EMF), http://wiki.eclipse.org/emf (2012)
32. Fleurey, F., Baudry, B., Muller, P., Traon, Y.: Qualifying input test data for model transformations. Softw. Syst. Model. (SoSym) **8**, 185–203 (2009)
33. Fleurey, F., Breton, E., Baudry, B., Nicolas, A., Jézéquel, J.-M.: Model-driven engineering for software migration in a large industrial context. In: Model Driven Engineering Languages and Systems (MoDELS), pp. 482–497. Springer, Berlin (2007)
34. Fleurey, F., Steel, J., Baudry, B.: Validation in model-driven engineering: testing model transformations. In: Model, Design and Validation (MoDeVa), pp. 29–40. IEEE (2004)
35. Gardner, T., Griffin, C., Koehler, J., Hauser, R.: A review of OMG MOF 2.0 query/views/transformations submissions and recommendations towards the final standard. In: MetaModelling for MDA, Workshop, pp. 178–197 (2003)
36. Giese, H., Hildebrandt, S., Neumann, S.: Model synchronization at work: keeping SysML and AUTOSAR models consistent. In: Graph Transformations and Model-Driven Engineering, Vol. 5765, pp. 555–579. Springer, Berlin (2010)
37. Gogolla, M., Vallecillo, A.: Tractable model transformation testing. In: European Conference on Modelling Foundations and Applications (ECMFA), pp. 221–235 (2011)
38. Haschemi, S.: Model transformations to satisfy all-configurations-transitions on statecharts. In: Model-Driven Engineering, Verification and Validation (MODEVVA) (2009)
39. Hausmann, J.H.: Metamodeling relations-relating metamodels. In: Metamodelling for MDA, pp. 147–161 (2003)
40. Hemel, Z., Kats, L.C., Groenewegen, D.M., Visser, E.: Code generation by model transformation: a case study in transformation modularity. In: Software and Systems Modeling (SoSyM), vol. 9, pp. 375–402. Springer, Berlin (2010)
41. IBM. IBM Rational Asset Manager (RAM) http://www-01.ibm.com/software/rational/products/ram/
42. IBM. IBM Rational Rhapsody, http://www.ibm.com/developerworks/downloads/r/rhapsodydeveloper/
43. iFEST. http://www.artemis-ifest.eu/home
44. IRISA. Kermeta, http://www.kermeta.org/ (2012)
45. IRISA. Metamodel Coverage Checker (MMCC). http://www.irisa.fr/triskell/Software/protos/MMCC (2012)
46. Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I.: ATL: A model transformation tool. In: Science of Computer Programming, vol. 72, pp. 31–39. Elsevier, Amsterdam (2008)
47. Kent, S., Smith, R.: The bidirectional mapping problem. Electron. Notes Theor. Comput. Sci. **82**(7), 151–165 (2003)
48. Kolovos, D., Paige, R., Polack, F.: A framework for composing modular and interoperable model management tasks. In: Model-Driven Tool and Process Integration Workshop (MDTPI), pp. 79–90. Berlin, Germany (2008)
49. König, B., Kozioura, V.: Augur 2-A new version of a tool for the analysis of graph transformation systems. In: Electronic Notes in Theoretical Computer Science (ENTCS), vol. 211, pp 201–210. Elsevier, Amsterdam (2008)
50. Küster, J., Abd-El-Razik, M.: Validation of model transformations-first experiences using a white box approach. In: Models in Software Engineering, pp. 193–204 (2007)
51. Mandelin, D., Kimelman, D., Yellin, D.: A Bayesian approach to diagram matching with application to architectural models. In: International Conference on Software Engineering (ICSE), pp. 222–231. Shanghai, China (2006)

52. Maskeri, G., Willans, J., Clark, T., Evans, A., Kent, S., Sammut, P.: A pattern based approach to defining translations between languages (2002)
53. McQuillan, J., Power, J.: A Survey of UML-based coverage criteria for software testing. Technical Report, Department of Computer Science, NUI Maynooth, Co. Kildare, Ireland (2005)
54. Mens, T., Van Gorp, P.: A taxonomy of model transformation. In: Electronic Notes in Theoretical Computer Science, vol. 152, pp. 125–142. Elsevier, Amsterdam (2006)
55. Mohagheghi, P., Dehlen, V.: Where is the proof?: a review of experiences from applying MDE in industry. In: European Conference on Model Driven Architecture-Foundations and Applications (ECMDA-FA), pp. 432–443. Springer, Berlin (2008)
56. Mottu, J., Baudry, B., Le Traon, Y.: Mutation analysis testing for model transformations. In: European Conference on Model Driven Architecture-Foundations and Applications (ECMDA-FA), pp. 376–390 (2006)
57. Narayanan, A., Karsai, G.: Verifying model transformations by structural correspondence. In: Electronic Communications of the European Association of Software Science and Technology (EASST), Vol. 10 (2008)
58. Nejati, S., Sabetzadeh, M., Chechik, M., Easterbrook, S., Zave, P.: Matching and merging of statecharts specifications. In: International Conference on Software Engineering (ICSE), pp. 54–64. Minneapolis, USA (2007)
59. Offutt, J., Abdurazik, A.: Generating tests from UML specifications. In: The Unified Modeling Language (UML), pp. 416–429. Springer, Berlin (1999)
60. OMG, O.: Meta Object Facility (MOF) Specification Version 1.4 (2002)
61. Ostrand, T., Balcer, M.: The category-partition method for specifying and generating functional tests. Commun. ACM **31**, 676–686 (1988)
62. Peterson, J.: Petri nets. In: ACM Computing Surveys (CSUR), vol. 9, pp. 223–252. ACM, New York (1977)
63. Pressman, R.S.: Software Engineering: A Practitioner's Approach, 7th edn. McGraw-Hill, New York (2009)
64. Rivera, J., Guerra, E., de Lara, J., Vallecillo, A.: Analyzing rule-based behavioral semantics of visual modeling languages with Maude. In: Software Language Engineering (SLE), pp. 54–73 (2009)
65. Rozenberg, G., Ehrig, H.: Handbook of Graph Grammars and Computing by Graph Transformation, vol. 1. World Scientific, Singapore (1999)
66. Salay, R., Wang, S., Suen, V.: Managing related models in vehicle control software development. In: Model Driven Engineering Languages and Systems (MoDELS), pp. 383–398. Springer, Berlin (2012)
67. Schätz, B., Holzl, F., Lundkvist, T.: Design-space exploration through constraint-based model-transformation. In: Engineering of Computer Based Systems (ECBS), pp. 173–182. Oxford, UK (2010)
68. Selim, G., Cordy, J., Dingel, J.: Analysis of model transformations. In: Technical Report 2012-592, School of Computing, Queen's University (2012)
69. Selim, G., Cordy, J., Dingel, J.: Model transformation testing: the state of the art. In: Analysis of Model Transformations (AMT), Innsbruck, Austria (2012)
70. Selim, G., Wang, S., Cordy, J., Dingel, J.: Model transformations for migrating legacy models: an industrial case study. In: European Conference on Modelling Foundations and Applications (ECMFA), pp. 90–101. Springer, Berlin (2012)
71. Sodius. MDWorkbench, http://www.mdworkbench.com/ (2012)
72. Soley, R., The OMG Staff: Model Driven Architecture. In OMG white paper, November (2000)
73. Steinberg, D., Budinsky, F., Merks, E., Paternostro, M.: EMF: Eclipse Modeling Framework, Chapter 5: Ecore Modeling Concepts. Addison-Wesley, Reading (2008)
74. Syriani, E.: Matters of model transformation. In: Technical Report, School of Computer Science, McGill University, SOCS-TR-2009.2, March (2009)
75. Teppola, S., Parviainen, P., Takalo, J.: Challenges in deployment of model driven development. In: International Conference on Software Engineering Advances (ICSEA), pp. 15–20. Porto, Portugal (2009)
76. Varró, D., Varró-Gyapay, S., Ehrig, H., Prange, U., Taentzer, G.: Termination analysis of model transformations by Petri Nets. In: International Conference on Graph Transformations (ICGT), pp. 260–274. Springer, Berlin (2006)
77. Wu, Y., Chen, M., Offutt, J.: UML-based integration testing for component-based software. In: The International Conference on COTS-Based Software Systems (ICCBSS), pp. 251–260 (2003)

Author Biographies



Gehan M. K. Selim received an M.Sc. from Cairo University (Faculty of Computers and Information) in Egypt and is currently a Ph.D. candidate in the School of Computing of Queen's University in Canada. Her research interests include model transformations, testing of model transformations, and formal verification of model transformations.

Shige Wang received his Ph.D. in Computer Science and Engineering from University of Michigan at Ann Arbor. He is currently working as a Senior Research Scientist at General Motors R&D in Warren, Michigan, USA. His research interests are in real-time and embedded systems and cyber-physical systems. Dr. Wang is an IEEE Senior Member.

James R. Cordy is Professor and past Director of the School of Computing at Queen's University in Kingston, ON, Canada. From 1995 to 2000 he was Vice President and Chief Research Scientist at Legasys Corporation, a software technology company specializing in legacy software system analysis and renovation. He has published more than 150 refereed contributions in software engineering, programming languages, and artificial intelligence. Dr. Cordy serves widely as member and chair of conferences and workshops in programming languages and software engineering, recently chairing ICSM 2011, SCAM 2012, WCRE 2013, and the 2012 Dagstuhl Workshop on Software Clone Management in Industrial Application. He is an ACM Distinguished Scientist, a Senior Member of the IEEE, and an IBM CAS Faculty Fellow.

Juergen Dingel received an M.Sc. from Berlin University of Technology in Germany and a Ph.D. in Computer Science from Carnegie Mellon University (2000). He is an Associate Professor in the School of Computing at Queen's University where he leads the Modeling and Analysis in Software Engineering group. His research interests include model-driven engineering, formal methods, and software engineering.