30th Annual INCOSE international symposium
Virtual Event
July 20 - 22, 2020

General Modeling Language Supporting Model Transformations of MBSE (Part 2)

Jiangmin Guo
Beijing Institute of Technology
5 South Zhongguancun Street, Haidian District, Beijing, China

Guoxin Wang
Beijing Institute of Technology
5 South Zhongguancun Street, Haidian District, Beijing, China

Jinzhi Lu
École Polytechnique Fédérale de Lausanne
Route Cantonale, 1015 Lausanne, Switzerland
jinzhi.lu@epfl.ch

Junda Ma
Beijing Institute of Technology
5 South Zhongguancun Street, Haidian District, Beijing, China

Martin Törngren
KHT Royal Institute of Technology
Brinellvägen 83

Abstract. With the increasing complexity of systems, model-based systems engineering (MBSE) has attracted increasing attention in the industry. MBSE formalizes the whole lifecycles of products using models based on systems engineering aiming to improve the development efficiency of complex systems. Traditionally, MBSE approaches require many modeling languages in each phase of the entire lifecycle. Different syntax between such languages leads to difficulty in supporting an integrated description of transformations between models and data. Thus, it is challenged to utilize a general language to describe model formalism and transformation for architecture-driven technology and code generation in one MBSE tool. In this paper, a multi-architecture modeling language called *Karma* (introduced in Paper Part 1) is proposed to support the model transformations including architecture-driven technology and code generation implementations in one modeling tool. Finally, from one auto-braking case of an autonomous-driving system, we find the availability of the *Karma* language supporting architecture-driven technology and code generation is verified.

Introduction

Model-based systems engineering (MBSE) is an emerging technology supporting complex system development, which has been widely favored in both academia and industry. MBSE formalizes requirements, design, analysis, verification, validation, and other specific development activities involved in the entire lifecycle based on a formal modeling approach (INCOSE Website 2012). MBSE aims to promote the development efficiency and system performance of complex systems, to improve communications among stakeholders and to enforce design automation.

Generally, each MBSE approach provides its formal language to support complex system development. Most of the languages are adopted to formalize the system artifacts and development without support of describing transformations between models, such as the transformations from requirement models to functional or architecture models. Generally, the model transformation technology can be

used to implement the conversions of model-to-model and model-to-text. Model transformations are widely used in software development which is a central concept in model-driven development approaches and providing a way for manipulating models automatically(Biehl 2010). But there are numerous independent model transformation languages available to support MBSE. Thus, system engineers participating in complex system development using MBSE are usually required to learn multiple languages and acquire the theoretical knowledge of software engineering and computer fundamentals.

This paper aims at extending from the first goal (as shown in the paper "General Modeling Language to Support Model-based System Engineering Formalisms (Part 1)" [1], which is a multi-architecture modeling language, *Karma* to support the entire MBSE formalisms across the entire lifecycle). The goal here specifically refers to a unified formal description of model transformations based on the *Karma* language. Currently, the *Karma* language is implemented in a Commercial-Off-The-Shelf (COTS) *MetaGraph*[2]. In the future, the *Karma* language is expected to publish as one open-source language specification. One solution is proposed to support model transformations including architecture-driven technology and code generation implementations in one multi-architecture modeling tool. The main contributions are shown as follows:

Supporting the formal expression utilized in model transformations for architecture-driven technology. *Karma* language supports not only the complete formalisms of MBSE, but also the expression required to define transformations for architecture-driven technology between models in one MBSE tool. The concept of architecture-driven is generally used in the field of software development(Thalheim & Jaakkola 2010). Here, architecture-driven technology refers to the model-to-model transformations where the source model is mapped into the target model instead of the text. Generally, top-down approaches are used when building systems where the whole system structure is defined first and subsystem implementations are added after that. So the top-level architecture models are built firstly, then subsystem models can be generated incrementally using model transformation. In this paper, such a transformation is called architecture-driven technology.

Supporting the formal expression utilized in model transformations for code generation. In addition to supporting the formal expressions of architecture-driven technology, *Karma* language also supports the formal expressions aiming to describe the transformation rules for code generation. Code generation refers to model-to-text transformations. Elements in the source model are mapped to target codes which can be identified by external tools. For example, with the use of the *Karma* language, a physical architecture model in the developed MBSE tool automatically generates target codes, documents, XML files, and other formal data structures.

The paper is organized as follows: First, related work is introduced. Then the formal expressions of architecture-driven technology and code generation supported by *Karma* language are demonstrated. Moreover, an auto-braking case in an autonomous driving system is used to verify the availability of the *Karma* language. Finally, the conclusion and future work are presented in the last section.

Related work

Owing to its unique capability to describe system lifecycles using models, MBSE has received increasing attention. Specifically, model transformations for architecture-driven technology within one tool and code generation across tools have been the core preoccupation of MBSE (Jean 2008). Researchers in this field have done comprehensive research on architecture-driven technology and automated code generation. For example, ATLAS Transformation Language (ATL) (Jouault & Kurtev

[1] Defined in Paper A "General Modeling Language to Support Model-based System Engineering Formalisms" of IS 2020. The first goal refers to formalizing the MBSE languages using *Karma* language based on a M3-M0 modeling framework. The second goal refers to integrated formalisms of the model transformations for architecture-driven technology and code generation.
[2] MetaGraph is a multi-architecture modeling tool developed by Z.K. fC http://www.zkhoneycomb.com/ .

2006) and GReAT languages (Partsch & Steinbrüggen 1983) support to formalize model transformations, while languages such as MERL (Pohjonen 2005) support automated code generation.

Many transformation languages are developed based on Query/View/Transformation (QVT) (ManagementGroup 2003) which is a standard set of model transformation languages defined by the Object Management Group (OMG), describing the process and method of transformation from a source model into a target model. ATL (Jouault et al. 2008) is a domain-specific language based on QVT utilized in the transformations between specified models. The ATL is implemented based on the Eclipse Modeling Framework (EMF), where meta-models and models are defined referring to a rule-based model transformation language. However, the disadvantage of the ATL is strongly dependent on the EMF framework and lies in its inability to verify the consistency of meta-models causing errors in model transformation (Grangel et al. 2007). GReAT is a model transformation language, utilized in model transformations based on meta-models in the Generic Modeling Environment (GME). GReAT supports transformations among multiple model diagrams (models belonging to various domains) based on a model graph generator. With pattern specifications, such diagram transformations are controlled to meet the specific requirements in the model transformation domain (Mens & van Gorp 2005). Hans proposed one transformation language for architecture-driven techniques (Syriani & Vangheluwe 2010). They developed one visualized model to formalize the transformation rules to transform more efficiency. Though these languages aim to formalize model transformations, meta-models are developed based on other modeling languages which means that developers need to know at least two languages for modeling and implementing the model transformations. The R&D cost for learning additional languages decreases the development efficiency using MBSE and becomes one of the important hinders to use MBSE in real industrial practices (Lu et al. 2018).

The OMG's Model-Driven Architecture (MDA) (Poole 2001) uses the Unified Modeling Language (UML) as a specification language for automated code generation. The UML is mainly utilized to describe programming concepts and implement code generations with the support of other modeling languages. For instance, the MetaEdit+ (Kelly & Tolvanen 2008) supports domain-specific modeling (DSM) based on UML and to implement code generations based on the MEAL language. Besides, it requires domain-specific modeling experts with a software engineering background to develop the code-generators themselves. Moreover, Architecture Analysis and Design Language (AADL) is an architecture description language proposed by the Society of Automotive Engineers (SAE) (Feiler et al. 2006). Mainly, ADDL is applied in the embedded system supporting a model-driven development and code generation in the entire lifecycle. Though these methods support self-configuration for implementing code generations, they do not support model transformations for architecture-driven technology in one modeling tool, so developers need to learn more than one language to realize whole transformations.

Based on the problems in existing methods, one MBSE multi-architecture formal language, *Karma*, is proposed to support the unified formal expression of MBSE and model transformations for architecture-driven technology and code generation. The detailed introduction of *Karma* language supporting MBSE formalisms is introduced in another paper, "General Modeling Language to Support Model-based System Engineering Formalisms (Part 1)". The *Karma* language integrates MBSE formalisms and descriptions of architecture-driven technology and code generation into one language. This improves model unification and development efficiency, as well as reducing the R&D cost and the stress arising from having to learn more than one language by developers for their works.

Implementing model transformations using *Karma* language

Overview

In this section, the workflows implementing model transformations for architecture-driven technology and code generation based on *Karma* language using *MetaGraph*, are first introduced. Afterward,

the workflows whereby *MetaGraph* implements the *Karma* language to support the model transformations are illustrated separately. Finally, the syntax of the *Karma* language is introduced.

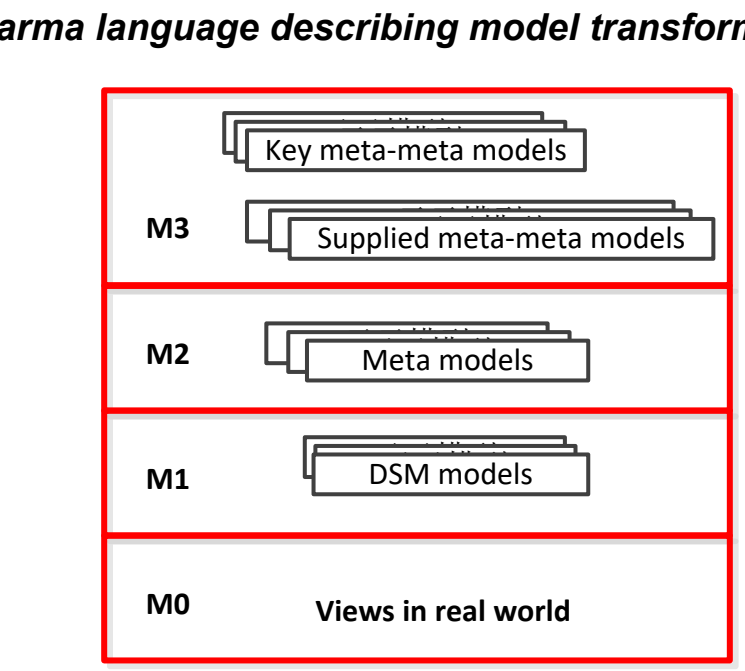


Figure 1. M0-M3 Domain-specific Modeling

***Karma* language implementing architecture-driven technology:** The *Karma* language first supports MBSE formalisms based on GOPPRR (meta-meta models including Graph, Object, Point, Property, Role, and Relationship) approach(Kelly & Tolvanen 2008). As the basics for architecture-driven technology and code generation, the GOPPRR approach formalizes system artifacts and development using the unified meta-meta models as shown in Figure 1.

- M3 (Meta-meta models): Meta-meta models are defined based on GOPPRR meta-modeling language. The key meta-meta models include a graph, object, relationship, role, property, and point to support definitions of meta-models. The complementary meta-meta models include relationships between such meta-meta models, such as *decomposite* to support meta-model development.
- M2 (Meta-models): Meta-models defined based on M3. These meta-models are used to build DSM models for formalizing architectures in different hierarchical layers. Meta-models are instantiations of meta-meta-models.
- M1 (DSM models): Models formalizing architectures.
- M0 (Referring to system views): Architectural views in the real world described by models.

Based on such meta-meta models, *Karma* language is used to define the formalisms of transformation rules between meta-models in *MetaGraph*, and between meta-models in *MetaGraph* and models (or codes) in other modeling tools.

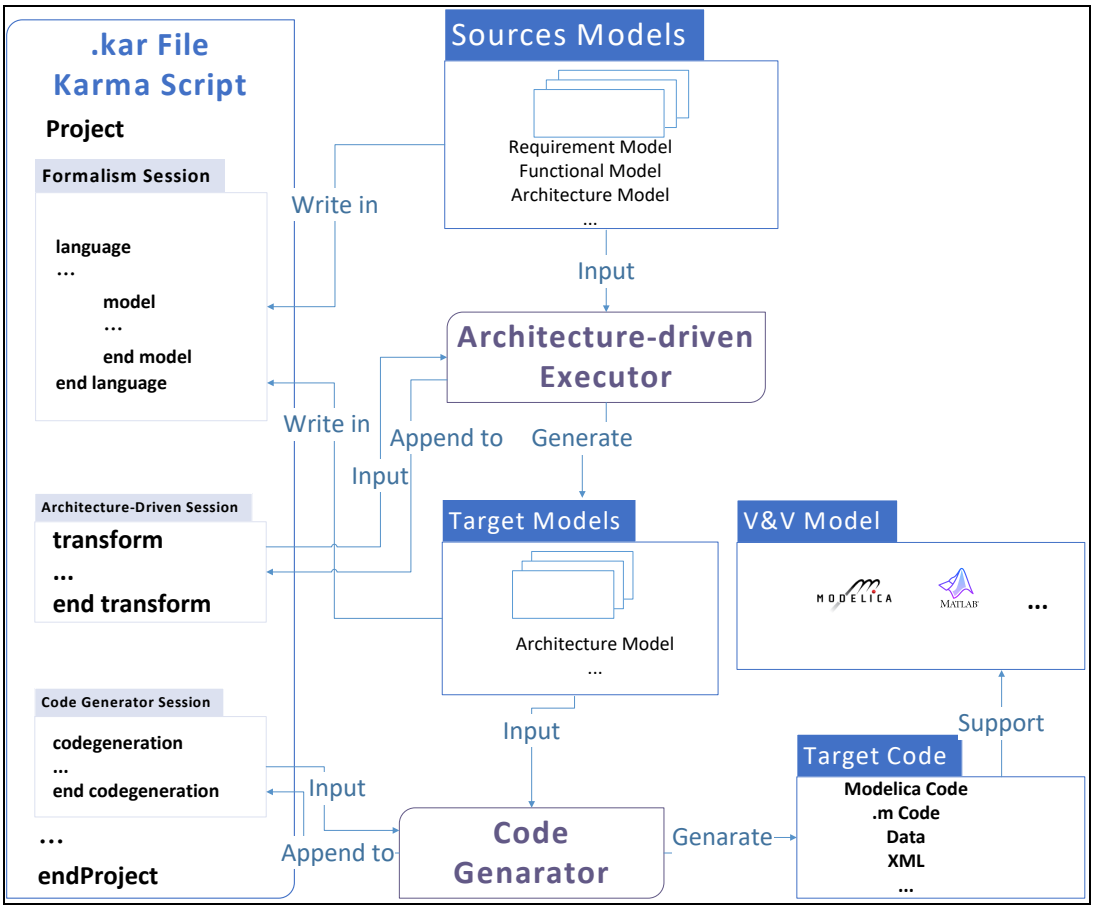


Figure 2. Workflows of *Karma* Language in *MetaGraph*

Figure 2 demonstrates the workflows of *Karma* language implementing these two types of model transformations in *MetaGraph*. Details are introduced as follows:

- The *.kar* file refers to one file describing *Karma* language, which includes all formal expressions of meta-meta models, meta-models and models, and formalisms of model transformations for architecture-driven technology and code generation. As seen in Figure 2, the three parts of the *.kar* file are introduced:

  (1) The formalism session refers to the formal description of MBSE. Based on the GOPPRR approach, *Karma* language supports the formal expression of models and describes all information related to the models. The language concept refers to one modeling language domain for meta-models, such as meta-models based on SysML are defined in the language session of SysML. Based on each language, models are formalized in the session of model concepts.

  (2) Architecture-driven session refers to the *Karma* language formalizing the model transformations for architecture-driven technology. *Karma* language has syntax to support the formal expression of model-to-model transformations using the architecture executor, which captures the information from the *Karma* files and implements the model transformation rules (details introduced in the next sub-section). Based on the rules, the executor creates target models in *MetaGraph* based on the source models described by *.kar* file.

  (3) Code generation session refers to the *Karma* language formalizing the code generation. The code generator captures this session in the *.kar* file and transforms the source models to target codes (or data). The syntax describes the code generation rules are introduced in the later section. The target codes are also used to realize automated verification & validation, e.g., generating Modelica and Simulink models automatically.

- During the architecture-driven process, source models refer to the formal models described by *Karma* language in *MetaGraph*, such as requirements models, functional models. Dependencies (Qamar 2013) between such models and target models are defined during the whole lifecycle, which is considered as the basics to design rules for architecture-driven technology(Mubeen et al. 2016). The target models refer to the created models after the model transformation, in which all information described by *Karma* languages is saved as the related sessions in the *.kar* files. The corresponding session can be changed from the front-end in the *MetaGraph*.

- During the code generation process, source models refer to the formal models described by the *Karma* language. The code generation is defined based on the code generation session and transforms source models to target codes. This *Karma* session can also be modified from the front-end in the *MetaGraph*. The target codes are the scripts generated by the code generator, such as MATLAB Scripts and XML reporting files.

- V&V Models refer to the simulation models for verification and validation. The target codes generated by the code generator support the development of simulation models. For example, the Simulink models are generated from MATLAB scripts.

Architecture-driven technology supported by the Karma language

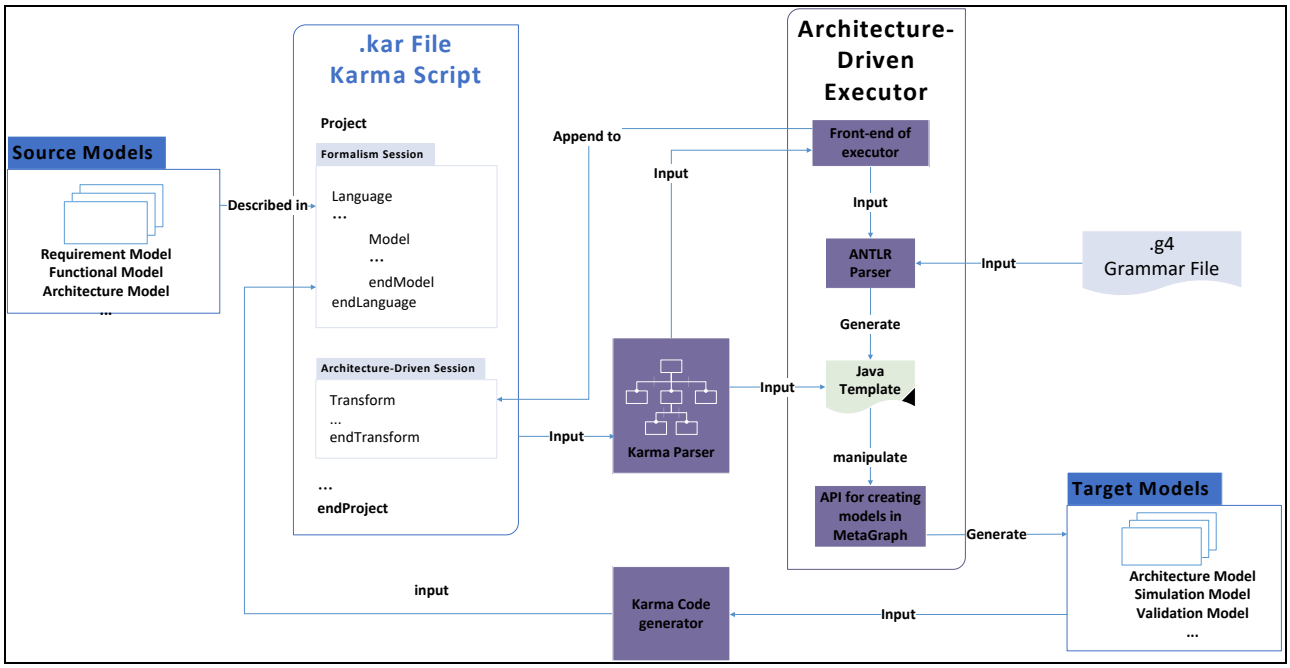


Figure 3. The Workflow of Architecture-driven Technology Supported by *Karma*

In this section, a workflow to implement the architecture-driven technology based on the *Karma* language is introduced as shown in Figure 3.

- Source models and the architecture-driven formalisms are described by the *Karma* scripts in the .kar file.

- A *Karma* parser is developed to capture the *Karma* language in the *.kar* file and generate an abstract syntax tree for these sessions. Then based on the abstract syntax tree, the architecture-driven session is input to the front-end of the executor where it also can be modified. The model sessions of source models in *.kar* files are interrupted by the parser and used to generate the Java templates to support model generations for architecture-driven technology.

- Moreover, a parser based on Another Tool for Language Recognition (ANTLR)[3] (Parr 2013) is developed to transform architecture-driven session into Java templates. The parser is defined based on a .g4 file referring to one syntax file of ANTLR for interrupting the architecture-driven session.
- The java templates are used for manipulating Application Programming Interfaces (API) into creating new models in *MetaGraph,* which are based on the model sessions obtained from the *Karma* parser.
- When the target models are created, the *Karma* code generator is used to save the new model sessions in the .kar files.

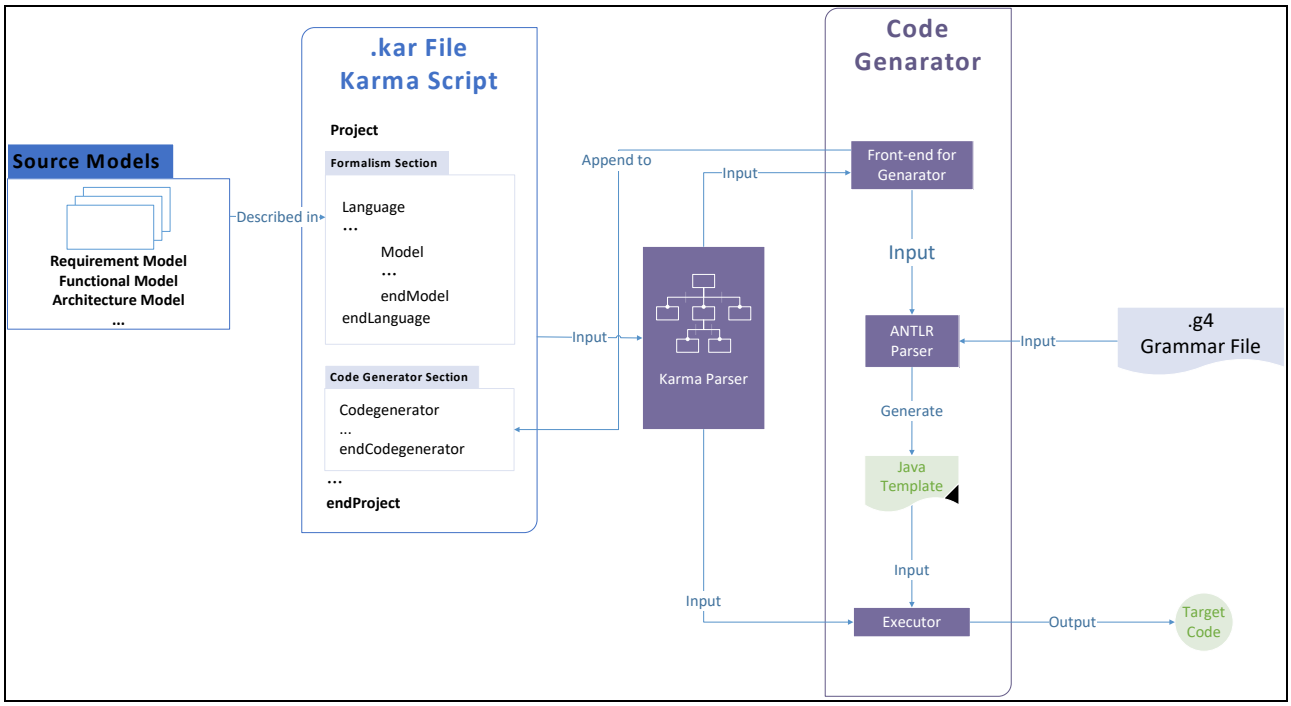


Figure 4. The Workflow of Code Generation Supported by *Karma* Language

Code generation supported by the Karma language

The code generator implements automated code generation based on the *Karma* language. It automatically transforms the source models in *MetaGraph* into target codes according to transformation rules defined in the code generator session. For example, Matlab scripts are generated through physical architecture models to create one new Simulink model introduced in the case study.

In Figure 4, source models are described in the model sessions of .kar files. The code generations are formalized in the *code-generation* sessions. The workflow is introduced as follows:

(1) ***Karma* language parsing.** Similar to the architecture-driven process, *Karma* parser interrupts the information about code generation into one abstract syntax tree. Afterward, based on the abstract syntax tree, the session of *code-generation* is input into the front-end of the code generator where this session can be modified as well. Moreover, the model sessions are input into the executor to provide enough information for executing code generation.

(2) **ANTLR parser interrupting the transformation rules based on the *Karma* language.**

[3] ANTLR is a predicated-LL(k) parser generator that handles lexers, parsers, and tree parsers. (Mens & van Gorp 2005)

A .g4 file is developed for the ANTLR parser in the code generator to interrupt the *code-generation* session and generate java templates for code-generation executions. Finally, the executor is used to implement java templates, to capture information from the model session in a .kar file and to generate the target codes.

Syntax of Karma language supporting model transformations

To realize the workflow introduced in the previous section, the concrete syntax of *Karma* language for describing model transformations for architecture-driven technology and code generation is presented in Table 1:

Table 1: Main Concrete Syntax of *Karma* Language for Model Transformations

Concrete Syntax	Description
General expression	
print(XXX);	Print the statements, which are used to output XXX value to the console. XXX represents attributes, string and so on.
fileName XXX write close	Statements are output to files. The results between this segment are written in files named XXX after running scripts.
Object/Relationship/Role . XXX{ }	Specify elements such as objects, relationships and role action, which are named XXX.

iterate Graph/Object/Relationship{
expression*
}
	Iterators are used to iterate over the currently pointed elements and performs the operations described in expression.

Declarations of Code generation	
codeGenerator XXX clause* end XXX	Define code generation fragments for the description of mapping rules of code generation. XXX represents the name of the code generator.
Declarations of Architecture drive	
architectureDriver XXX clause* end XXX	Model transformation is described in this section. The XXX is the name of the transformer.
XXX (*sourceMetaModel*,*targetMetaModel*);	Transformation rule is defined using this function, *sourceMetaModel* represent matching the meta type of elements and *targetMetaModel* represent the meta type of elements will be generated using this rule. One rule can be named XXX which is a legal string.
rule XXX *ruleClause** end XXX	Define the transform rule for architecture drive, any number of rules can be described in this segment. The *ruleClause* is a collection of the transformation rule above.
trans (*sourceElement*,*targetElement*,*rule*)	Apply one rule to the model, *sourceElement* represent existed elements in the source model. The *targetElement* represents elements in the target model which will be created after the *rule* is used.

Based on the concrete syntax above, the abstract syntax of *Karma* language for formalizing architecture-driven technology and code generation is defined. As illustrated, Figure 5 demonstrates the *Karma* session for code generation, while Figure 6 presents the *Karma* language sessions for implementing the architecture-driven technology.

```
codeGenerator TestGenerate %Define one generator named TestGenerate
fileName "TestCase.txt" write %Write the following results into the file
  %Declare two variables which type are string
  String Symple1 = " \"";
  String ModeleName = "Test";
  /*Foreach Objects*/
  for i = 1:number(Object)
    print(valueof(this.Function)); %Function is a property of Object
    print(Symple1); %Print the variable Symple1's value
    print(valueof(this.Path)); %Output the value of property "Path"
    print(Symple1, ModeleName,valueof(this.Name),Symple1;end="\n");
  end
  /*Foreach all Relationships*/
  for i = 1:number(Relationship)
    print(valueof(this.Function));
    %Foreach objects which are connected to the relationship
    for a = 1:number(Object)
      print(valueof(this.Name), end="\n");
    end
  end
close
end_TestGenerate
```

Figure 5. Example of the Code Generation Session

```
/*Transform the source model into the target model*/
architectureDriver TestTrans %Define a Transformer called TestTrans
%Declare one existed model which is regard as source model
model SourceModel = model.mSystemName;
%Create one model represents as output which is the instance of one meta graph
model TargetModel = model.m2SystemName instanceof metaGraph.mm2SystemName;
  %The section for defining transformation rules
  rule Transform1
    %This rule represents that when one instance of source object is find,
    %then one instance of target object is created in specific graph
    trans_object(
    metaObject.source_object_name, metaObject.target_object_name);
    %It represents the rule creating one relationship
    trans_relationship(
    metaRelationship.source_rel_name, metaRelationship.target_rel_name);
  end Transform1
  %Foreach Objects
  iterate Object(this.parent = SourceModel){
    %Transform the object into the target object in TargetModel
    trans(this,TargetModel.Object(),trans_object);
  }
  %Foreach Relationships
  iterate Relationship(this.parent = SourceModel){
    %Transform the relationship into the target relationship in TargetModel
    trans(this,TargetModel.Relationship(),trans_relationship);
  }
end_Testmodel
```

Figure 6. Example of the Architecture-driven Session

Case study and evaluation

In this section, an auto-braking case of an autonomous-driving system is adopted to validate the availability of the *Karma* language supporting model transformations. The case is developed based on the literature (Lu et al. 2016) involved two vehicles in which the auto-braking controllers in the second vehicle were verified through simulations using Simulink. Figure 7 describes the complete workflow of the *Karma* language supporting architecture-driven technology in *MetaGraph* and code generation in which models are transformed into M files for creating Simulink models as shown in (Figure 7-F).

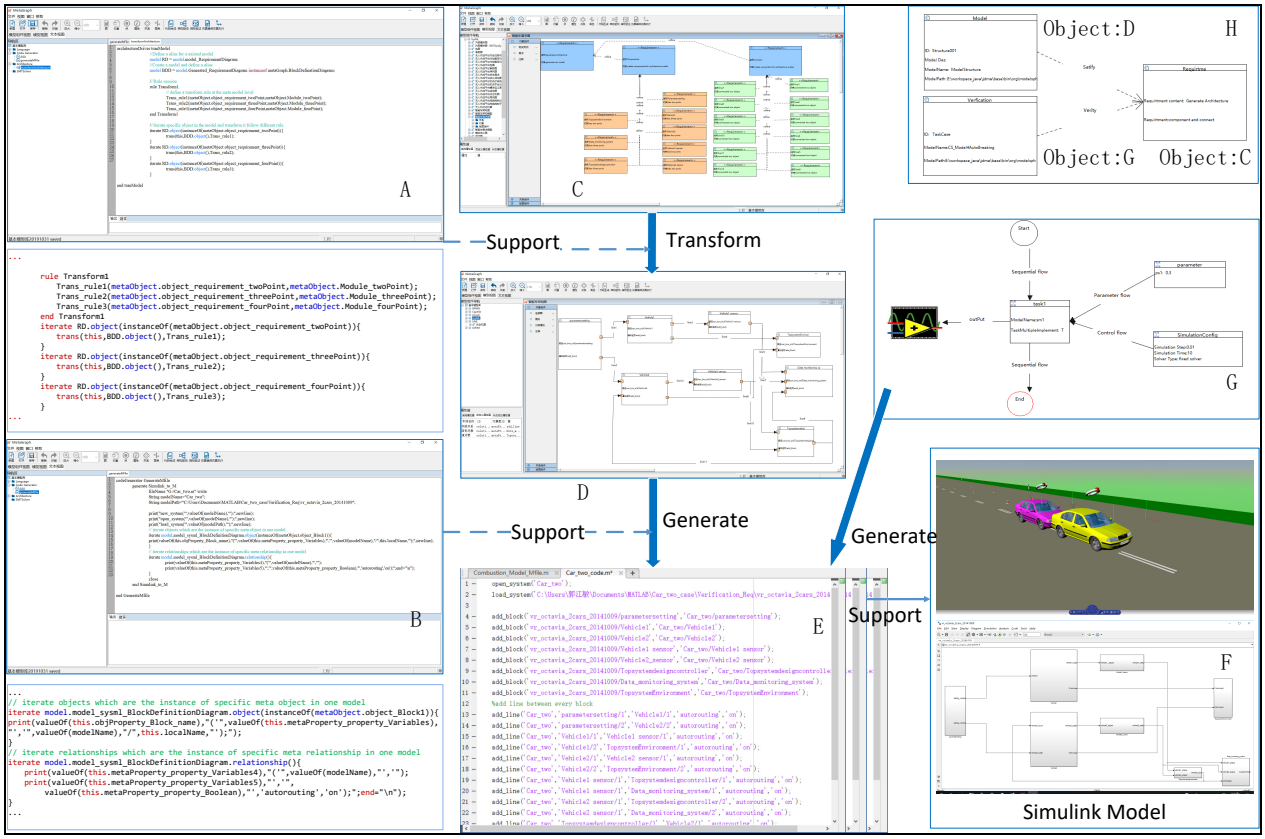


Figure 7. Automated Verification of Auto-braking Case Implemented by *Karma* Language

The complete four graphs are defined in Part 1[4] which are overview graph (Figure 7-H), requirement graph (Figure 7-C), the model structure graph (Figure 7-D), and the validation and verification graph (Figure 7-G). The overview graph represents the relationships between the three graphs. Object:C refers to one requirement object decomposed into a requirement graph (Figure 7-C). There are three types of meta-models to describe requirements: 1) The blue one refers to one general requirement object which describes general requirement, such as a requirement for simulation, creating models, etc. 2) The red one refers to the requirement to add a block during creating the models based on model structure graph. 3) The green one refers to the requirement to add relationships between objects. Therefore, the requirement graph describes the requirement for creating one architecture model is associated with requirements for adding objects and relationships.

The *Object:D* in the entire graph refers to one object for model structure decomposed into the model structure graph. The model structure graph describes the structures of the Simulink model. Object with different numbers of points refers to different meta-models constructing the model structures.

[4] Defined in Paper A, General Modeling Language to Support Model-based System Engineering Formalisms IS 2020. It refers to MBSE formalisms using *Karma* language based on a M3-M0 modeling framework.

The *Object: G* refers to one verification &validation graph which represents one sequenced task referring to one simulation. Each task is associated with results, parameters and simulation configurations.

The model developed based on the requirement graph is adopted as the source model for architecture-driven technology to generate a new model based on the model structure graph. The models developed based on the model structure and verification &validation graphs are used as source models for code generation. Such model transformations for the auto-braking cases are implemented by *Karma* language using *MetaGraph*. First, the model based on the requirement graph is created in *MetaGraph* manually where the requirement objects are defined. Then for supporting the architecture-driven technology, *Karma* language is used to formalize the transformation from the requirement graph to a model structure graph based on Algorithm 1. In the requirement model, all the elements are queried. When one requirement object is found, its related component in the model structure graph is added named as a defined property called *RequiredCompName* by applying the rule called *rule4Object*. Moreover, when one requirement object for creating *Relationship* is found, one relationship in the model structure graph is added named as a defined property called *RequiredRelationshipName* by applying the rule called *rule4Rel*. Moreover, its input and output are defined as object properties. As shown in (Figure 7-A), the transformation rules are defined in the front-end of the executor. Then the new model is created automatically based on the information captured from the model which is developed based on the requirement graph.

Algorithm 1 Generate One Model Based on Model Structure Graph Using Architecture Driven

1. **foreach elements in Requirement model** % all elements in Requirement model
2. **if** <Reqobject4Object is found> **then** % if one required object is found
3. trans(Reqobject4Object.RequiredCompName, ArcModel.newObject, rule4Object); % create one new object in ArcModel by applying the rule
4. **end if**
5. **if** <ReqObject4Relationship is found> **then** % if one required relationship is found
6. trans(ReqObject4Relationship.RequiredRelationshipName,ArcModel.newRelationship (ReqObject4Relationship.inRoleName, ReqOject4Relationship. outRoleName), rule4Rel);
   % create one new relationship in ArcModel
7. **end if**
8. **end for**

Algorithm 2 Generating M Files Using Code Generation

1. **for elements in ArcModel**
2. **if** <CompObject is found> **then** % if one required Component object is found
3. print ("add_block" ,object's property); % print add_block function in M files
4. **end if**
5. **if** <RelationshipObject is found> **then** % if one required relationship in ArcModel is found
6. <find startPoint and endPoint which are bound with the Relationship through Roles>
7. print ("add line", relationship's property (line's name in Simulink), endPoint'property (Start point of the line), startPoint'property (End point of the line)); % print add_line function in M files
8. **end if**
9. **end for**

Moreover, the *Karma* language is used to formalize the code generation from models based on model structure and verification &validation graphs to M files to support automated Simulink model creation and execution based on Algorithm 2. All the elements in *ArcModel* are queried and when required objects and relationships are found, *add_block* and *add_line* functions are printed in one M file. In Figure 7-B, the transformation rules for code generation are defined in the front-end of the code-generator. After the code generation, M files are generated automatically as shown in Figure 7-E, which are used to create Simulink models as shown in Figure 7-F.

From the case study, the availability of *Karma* language supporting architecture-driven technology and code generation is validated. Using the *Karma* language, the architecture driver and code generator are implemented for model transformations which represent the dependencies and traceability between different system aspects described by models. However, the internal validity is still not enough to validate the completeness of this language. More engineering practices are required to promote the syntax of the *Karma* language for better performance.

This paper refines our earlier case with an integrated description language called *Karma.* In our previous paper(Lu et al. 2016), the case of a vehicle auto-braking system is developed to illustrate the model-driven tool-integration framework. The Requirement models and the top structure model are built in MetaEdit and the M files are generated using MERL which is the description language for code generation supported by MetaEdit. However, some models such as the top structure model are built manually because of the lack of architecture-driven technology. Instead of adopting a strategy of introducing external language such as OSLC, we develop an integrated description language to formalize the whole life cycle models and model transformation including architecture-driven technology and code generation.

Compared with existing approaches, the *Karma* language integrates architecture-driven technology and code generation with MBSE formalisms, which avoids the developers learn more languages when they implement MBSE in their real work. Moreover, the unified formalisms enable the creation of knowledge for systems aspects including not only formalisms but also dependency and traceability for architecture-driven technology and code generation. Besides, if such unified descriptions are defined, one integrated compiler for language interrupting is possible to be developed. This is one potential good way to promote interoperability among MBSE tools and separated modeling environments, solvers, and processors for architecture-driven technology and code generation. This is one future goal for MBSE tool development (Beihoff et al. 2010). Finally, the integrated descriptions are the potential to define logic formalisms among system formalisms, dependencies, and traceability. At the same time, *Karma* language has the potentials to extend further capabilities for property analysis, constraint description and satisfiability of logical formulas.

Conclusion

In this paper, a multi-architecture modeling language, *Karma* is introduced aiming at formalizing MBSE formalisms, architecture-driven technology and code generation. The integrated descriptions promote interoperability between different MBSE models and different architecture driver and code generator. Currently, the workflow is implemented in a developed multi-architecture modeling tool, *MetaGraph* to explain how *Karma* language is implemented for architecture-driven technology and code generation. Then syntax of *Karma* language is introduced to illustrate how the *Karma* language formalizes these model transformations. Finally, based on one auto-braking case, the availability of the *Karma* language is validated. In the future, some work will be done:

- A language checker will be developed first to check the grammars in the *MetaGraph*.
- The completeness of the architecture driven and code generation sessions will be externally validated. Then more syntax will be developed based on ATL to support definitions of more complex transformation rules.

- The *Karma* language aims to be open with a specification and one corresponding compiler.

References

Beihoff, B, Friedenthal, S, Kemp, D, Nichols, D, Oster, C, Peredis, C, Stoewer, H & Wade, J 2010, 'A World in a Grain', *Science*, vol. 327, no. 5970, pp. 1183–1183, viewed <https://www.sciencemag.org/lookup/doi/10.1126/science.327.5970.1183-d>.

Biehl, M 2010, 'Literature study on model transformations', *Royal Institute of Technology, Tech. Rep. ISRN/KTH/MMK*, vol. 291.

Feiler, PH, Gluch, David P, Hudak, J, Gluch, D. P. & Hudak, JJ 2006, *The Architecture Analysis & Design Language (AADL): An Introduction*, Carnegie-Mellon Univ Pittsburgh PA Software Engineering Inst, viewed <http://www.sei.cmu.edu/library/abstracts/reports/06tn011.cfm>.

Grangel, R, Salem, R Ben, Bourey, J-P, Daclin, N & Ducq, Y 2007, 'Transforming GRAI Extended Actigrams into UML Activity Diagrams: a First Step to Model Driven Interoperability', *Enterprise Interoperability II*, Springer London, London, pp. 447–458, viewed <http://dx.doi.org/10.1007/978-1-84628-858-6_48>.

INCOSE Website 2012, 'Website des International Council on Systems Engineering', *Guide for Writing Requirements", INCOSE*, viewed <http://www.incose.org/>.

Jean, B 2008, *Model Driven Engineering Languages and Systems*, K Czarnecki, I Ober, J-M Bruel, A Uhl & M Völter (eds), *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, Berlin, Heidelberg, viewed <http://link.springer.com/10.1007/978-3-540-87875-9>.

Jouault, F & Kurtev, I 2006, 'Transforming Models with ATL', *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, Springer, pp. 128–138, viewed <http://sosym.dcs.kcl.ac.uk/events/mtip05/submissions/jouault_kurtev__transforming_models_with_atl.pdf>.

——— 2008, 'ATL: A model transformation tool', *Science of Computer Programming*, vol. 72, no. 1–2, Elsevier, pp. 31–39, viewed <http://linkinghub.elsevier.com/retrieve/pii/S0167642308000439>.

Kelly, S & Tolvanen, J-P 2008, *Domain-Specific Modeling: Enabling Full Code Generation*, Wiley-IEEE., Wiley-IEEE Computer Society Press.

Lu, J, Chen, D, Törngren, M & Loiret, F 2016, 'A model-driven and tool-integration framework for whole vehicle co-simulation environments', *8th European Congress on Embedded Real Time Software and Systems (ERTS 2016)*.

——— 2018, 'MBSE Applicability Analysis in Chinese Industry', *INCOSE International Symposium*, vol. 28, no. 1, pp. 1037–1051, viewed <http://doi.wiley.com/10.1002/j.2334-5837.2018.00532.x>.

Mens, T & van Gorp, P 2005, 'A taxonomy of model transformation and its application to graph transformation technology', *Proc. Int'l Workshop GraMoT*.

Mubeen, S, Ashjaei, M, Nolte, T, Lundback, J, Galnander, M & Lundback, K-L 2016, 'Modeling of End-to-End Resource Reservations in Component-Based Vehicular Embedded Systems', *2016 42th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, IEEE, pp. 283–292, viewed <http://ieeexplore.ieee.org/document/7592809/>.

Parr, T 2013, *The definitive ANTLR 4 reference*, Pragmatic Bookshelf.

Partsch, H & Steinbrüggen, R 1983, 'Program Transformation Systems', *ACM Computing Surveys (CSUR)*, vol. 15, no. 3, IBM, pp. 199–236, viewed <http://ieeexplore.ieee.org/document/5386627/>.

Pohjonen, R 2005, 'LNCS 3676 - Metamodeling Made Easy – MetaEdit+ (Tool Demonstration)', in R Glück & M Lowry (eds), *Lncs*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 442–446.

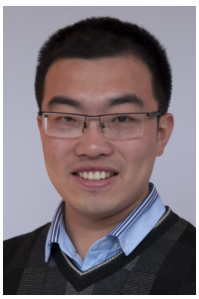
Poole, JD 2001, 'Model-Driven Architecture : Vision , Standards And Emerging Technologies', *Workshop on Metamodeling and Adaptive Object Models, ECOOP*, pp. 1–15.

Qamar, A 2013, *Model and dependency management in mechatronic design*, KTH Royal Institute of Technology.
Syriani, E & Vangheluwe, H 2010, 'De- / Re-constructing Model Transformation Languages', *Electronic Communications of the EASST*, vol. 29.
Thalheim, B & Jaakkola, H 2010, 'Architecture-Driven Modelling Methodologies', *Frontiers in Artificial Intelligence and Applications*, no. 225, pp. 97–116, viewed <http://www.sei.cmu.edu/architecture/start/moderndefs.cfm>.

Biography

Jiangmin Guo is a master student at Beijing Institute of Technology, majoring in model-based system engineering(MBSE). His research interest is code generation and model transformation for architecture driven technology in MBSE tool development.

Guoxin Wang, is an associate professor and director of Industrial Engineering Institute at Beijing Institute of Technology. Meanwhile, he also a senior fellow of Chinese Mechanical Engineering Society and China Graphics Society. His research interests include Reconfigurable Manufacturing System and Knowledge based Engineering.



Jinzhi Lu, a research scientist at EPFL SCI-STI-DK. His research interests are MBSE tool-chain design and MBSE enterprise transitioning. He is CSEP (04944), a senior member of China Council on Systems Engineering (CCOSE), co-founder of MBSEWiki in Chinese, TC member in IEEE SMC MBSE Group, MBSE consultant in ANSYS China, PC members of INCOSE IS, IEEE SOSE and worldcist conferences; He is also the reviewers of IEEE ACCESS and Information & Communications Technology Express.

Junda Ma, a Ph.D. student at Beijing Institute of Technology, majoring in model-based system engineering (MBSE). His research interest is general modeling language in MBSE tool development, which supports model formalisms.

Martin Törngren has been a Professor in Embedded Control Systems at the Mechatronics division of the KTH Department of Machine Design since 2002. He has particular interest in Cyber-Physical Systems, model based engineering, architectural design, systems integration, and co-design of control applications and embedded systems. He has authored/co-authored more than 100 peer reviewed publications, and also been in charge of developing and leading graduate and continued education courses.