Available online at www.sciencedirect.com

ScienceDirect

IFAC PapersOnline CONFERENCE PAPER ARCHIVE

IFAC PapersOnLine 55-10 (2022) 3004–3009

# Code Generation Approach Supporting Complex System Modeling based on Graph Pattern Matching

**Jie Ding*. Jinzhi Lu**. Guoxin Wang*, Junda Ma*, Dimitris Kiritsis**, Yan Yan***

** Beijing Institute of Technology, Beijing 100081, China*

*** EPFL - École Polytechnique Fédérale de Lausanne, Lausanne, 1015, Switzerland*

*(e-mail: jinzhi.lu@epfl.ch)*

**Abstract**: Code generation is an effective way to drive the complex system development in model-based systems engineering. Currently, different code generators are developed for different modeling languages to deal with the development of systems with multi-domain. There are a lack of unified code generation approaches for multi-domain heterogeneous models. In addition, existing methods lack the ability to flexibly query and transform complex model structures to the target code, resulting in poor transformation efficiency. To solve the above problems, this paper proposes a unified approach which supports the code generation of heterogeneous models with complex model structure. First, The KARMA language based on GOPPRR-E meta-modeling approach is used for the unified formalism of models built by different modeling languages. Second, the code generation approach based on graph pattern matching is proposed to realize the query and transformation of complex model structures. Then, the syntax for code generation is integrated into KARMA and a compiler for code generation is developed. Finally, a case of unmanned vehicle system is taken to validate the effectiveness of the proposed approach.



*Keywords*: Code generation, MBSE, Model-driven, GOPPRR, Meta-modeling, KARMA language

## 1. INTRODUCTION

With the rapid growth of system complexity, Model-based Systems Engineering (MBSE) is increasingly applied to the design of complex systems. MBSE uses formal models to describe the requirements, design, analysis, verification and validation activities in the whole life cycle of system development (Hart, 2012). Formal system models are used as the single source of truth to drive the development of system thus improve the efficiency of the R&D process.

Code generation is one of the key technologies in MBSE, also known as model-to-text (M2T) transformation (Sebastián et al., 2020). It obtains information from source models through model queries, and then transforms the obtained information to target code or text (such as XML). The Automatic or semi-automatic transformation from system models to other heterogeneous data files can be realized by code generation, thus improving the traceability and consistency of data between system models and other data files in the design process. In addition, through code generation, system models can be transformed into preliminary detailed-design models (code, simulators, test cases, etc.) to improve design efficiency.

However, the design process of complex systems is a multi-domain process. When using the MBSE method for system design, models are created from different perspectives, which correspond to different domains. Models in different domains are commonly constructed by different modeling languages. For example, UML (OMG, 2017) is a software modeling language can be used to construct requirements, logical models for software development, and BPMN (OMG, 2011) is used for modeling business process. Different modeling languages may be described in different data formats. Therefore, there are different methods and tools to support the code generation of models in heterogeneous formats. Non-uniform methods and tools are a serious drain on resources in the R&D process.

To solve the problem above, this paper proposes a unified approach which supports the formalism of multi-domain models and the code generation of complex model structure. This approach makes use of the KARMA language based on GOPPRR-E meta-modeling approach to uniformly describe system models in different domains. Based on the unified description, the graph pattern matching approach is used to support code generation of complex model structures. Then, the KARMA syntax is extended to formalize code generation based on graph pattern matching. Finally, the compiler for the code generation of KARMA models is developed.

The rest of this paper is organized as follows. In Section 2, the related works about the code generation method of multi-domain models are introduced. Section 3 presents the proposed solution which based on the GOPPRR-E meta-modeling approach and graph pattern matching. Then Section 4 describes one case to validate the proposed approach. Finally, we offer the conclusions and future works in Section 5.

## 2. RELATED WORKS

From literature review, there are mainly two ways to realize the code generation of different domain models, one is


10.1016/j.ifacol.2022.10.189

customized code generators and the other is code generation languages.

UJECTOR is a customized generator that supports the generation of executable Java code from UML class diagrams, sequence diagrams and activity diagrams (Usman et al., 2009). Parada et al. (2011) proposed to generate skeleton of Java code from a UML class diagram using GenCode. However, only static information is used in the code generation. The modeling tool MagicDraw (Magic, N, 2016) has an embedded document generator, a Java generator and a C++ generator, which are used to support the code generation of SysML (OMG, 2017) and UML models. Customized code generators are highly automatically. However, for different modeling languages and development purposes, different generators need to be developed and integrated, resulting in poor scalability.

Acceleo, Jet and Xpand (The Eclipse Foundation, 2021) are code generation languages based on EMF (Eclipse Modeling Framework), using the Ecore meta-modeling approach. There are five basic concepts (EPackage, EStructuralFeature, EClass, EReference and EAttribute) in Ecore to support the description of different modeling languages (Steinberg et al., 2008). Acceleo, Jet and Xpand are template-based code generation languages. They realize queries on model elements in an imperative way that is difficult to describe queries on complex model structure. The MERL (MetaEdit Reporting Language) is proposed based on GOPPRR meta-modeling approach (Kelly et al., 2008). According to the comparison of Kern et al. (2011), The GOPPRR approach has richer expression ability than Ecore. MERL uses imperative navigation statements for model query, and only one model element can be obtained in one navigation, which requires to describe multiple navigations to obtain the complex model structure with multiple elements. Lu et al. (2020) proposed the textual modeling language KARMA based on GOPPRR-E meta-modeling approach to realize the unified formalism of multi-domain system models. However, KARMA lacks the ability to describe the code generation of system models currently.

## 3. CODE GENERATION METHOD FOR MULTI-DOMAIN MODELS

### *3.1 Uified formalism of models based on GOPPRR-E*

To realize the code generation of multi-domain system models, the KARMA language based on GOPPRR-E meta-modeling approach is used to uniformly formalize the heterogeneous models. The MOF framework (OMG, 2000) is the basis of GOPPRR-E meta-modeling approach.

As shown in Figure 1, MOF consists of four layers: M3-M0. The M3 layer refers to meta–meta models which are basic elements used to compose meta models. The M2 layer refers to the meta model layer, which is used to define models in M1 layer. Modeling languages are defined based on meta models. The M1 layer represents the model layer. Models are instances of meta models and enable to support system development. The M0 layer refers to the perspective of real systems, which is abstractly described by models in M1 layer.

The M3 layer in the GOPPRR-E meta-modeling approach has six meta-meta models: Graph, Object, Point, Relationship, Property and Role. Graph refers to a composition of Objects and their Relationships. Object refers to one entity in a Graph. Point refers to one port in Objects and cannot exist alone. Relationship refers to the association between Objects, which connected to Objects through Roles. Role represents the end of the Relationship. Roles bind with Objects or Points in Objects. Property refers to the attribute of the five other meta–meta models.

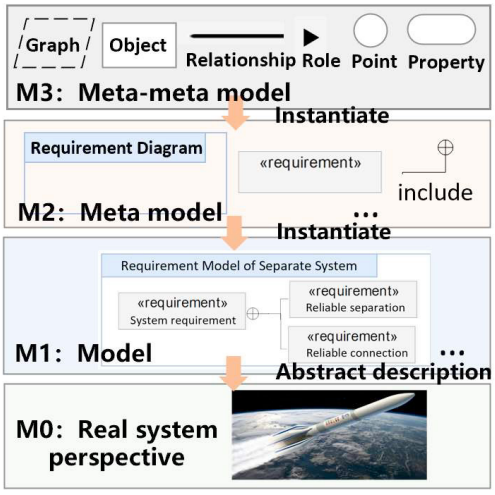


Figure 1 The GOPPRR-E meta-modeling approach based on MOF

An instance of a GOPPRR meta-meta model is a GOPPRR meta model, and an instance of a GOPPRR meta model is a GOPPRR instance. Specially, an instance of a Graph meta model is called as model. In addition, Extra rules (Extension) are used define the relation between six meta-meta models. The Extension includes the “include” relation between Graph meta models and Object and Relationship meta models, the “attach” relation between Point meta models and Object meta models, the “bind” relation between Role meta models and Point or Object meta models.

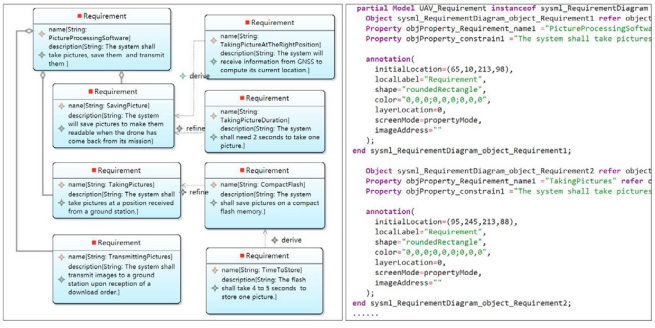


Figure 2 The Textual and Graphical KARMA Model

Based on GOPPRR-E meta-modeling approach, the semantic modeling language KARMA is developed (Lu et al., 2020), which supports the unified expression of different modeling languages such as SysML and UML. Moreover, a modeling tool called MetaGraph 2.0 (http://www.zkhoneycomb.com/) supports KARMA is developed, which supports modeling of GOPPRR meta models and instances both in textual and graphical form. As shown in Figure 2, the created graphical

model is automatically synchronized to the text syntax described by KARMA.

### *3.2 Code generation based on graph pattern matching*

On the basis of using KARMA to express the heterogeneous system models uniformly, this section proposes a code generation approach of KARMA models. First, the code generation process of KARMA models is introduced. Then, we proposed an approach based on graph pattern matching to realize the code generation process.

(1) The code generation process of KARMA models

The code generation process of KARMA models can be divided into two stages: model query and transformation. As shown in Figure 3, model queries are the process of querying and obtaining the structure of specified model elements in the source model to be transformed. We call the structure of specified model elements as model pattern (MP), the specific description of MP is as in (1):

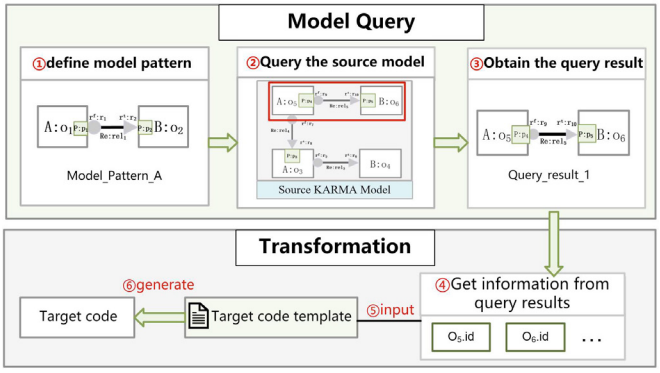


Figure 3 The code generation process of KARMA models

$$MP = (MP_S, MP_C ?) \quad (1)$$

MP consists of $MP_S$ (model pattern structure) and $MP_C$ (model pattern constraints). $MP_S$ is used to describe the model element instances (EI) that need to be queried and relations between EI. EI includes instances of Object, Relationship, Point, and Role (OPRR). The relations between EI includes the "include", "attach" and "bind". $MP_C$ is used to add constraints to the value of the Property instances and inherent information owned by the OPRR instances, for example, ID is one of the inherent information. Every OPPR instances uses ID as its unique identifier. The "?" in (1) means that $MP_C$ is not necessary.

Transformation is the process of transforming the information of obtained model elements into target code. As shown in Figure 3, the target code is generated by inputting the information of model elements into the target code template, which is a commonly used method and is not the focus of this paper. Model queries are the basis of transformation, and MP-based queries can obtain complex model structures with multiple elements in one query, enabling the generation of code or text with richer structure. This paper adopts the approach of graph pattern matching and use the graph matching engine to realize the MP-based model query.

(2) Model Query Based on Graph Pattern Matching

To realize the model query of KARMA models based on graph pattern matching, a formal definition of the KARMA model is given based on graph theory. In graph theory, a graph consists of a set of vertices and edges. In different scenarios, graphs can be expressed in different forms, such as directed graphs, attribute graphs and labeled graphs (Grunske et al., 2005). The relations between KARMA model elements are directional, and model elements have types and self-defined properties. Therefore, we use the directed, labeled graph with attributes to define the elements of KARMA model. The definition of a directed, labeled graph (G) with attributes (Champin et al., 1999) is as in (2). In addition, Attributes and attributes' values can be defined for G, v and e.

$$G = (V, E, source, target, label) \quad (2)$$

- V is a finite set of vertices (v) , $v \in V$.
- E is a finite set of edges. Each edge (e) has direction that points from the source vertex($v_s$) to the target vertex($v_t$).
- source: $e \rightarrow v_s$ and target: $v_t \rightarrow e$ are two mapping functions used to describe the connection between $v_s$, $v_t$ and e.
- The label represents the labeled mapping function which assigns type labels to all vertices V and edges E in G.

To describe KARMA models using G, we give an extension for concepts in G. Vertices are divided into Object vertices ($V_O$), Point vertices ($V_P$) and Role vertices ($V_R$). Edges are divided into Relationship edges ($E_R$), Bind edges ($E_B$), and Attach edges ($E_A$). The definition of GOPPRR instances and their relations based on graph theory is as follows:

- *attrs= (attrType, attrValue)*. The attrs is used to express the Property instance and inherent information owned by OPRR instances. When attrs is used to describe Property instances, the attrType refers to the Property meta model, otherwise refers to the type of inherent information. The attrValue refers to the value of attrs.
- KARMA model: $G_K$= *(gtype, V, E, attrs)*. The gtype refers to the type of $G_K$ which is the Graph meta model.
- Object instance: $v_O$= *(votype, attrs)*, $v_O \in V_O$. The votype refers to the type of $v_O$ which is the Object meta model.
- Point instance: $v_P$= *(vptype, attrs)*, $v_P \in V_P$. The vptype refers to the type of $v_P$ which is the Point meta model.
- Role instance: $v_R$= *(vrtype, attrs)*, $v_R \in V_R$. The vrtype refers to the type of $v_R$ which is the Role meta model.
- Relationship instance: $e_R$=*(etype, src, dst, attrs)*, $e_R \in E_R$. The etype represents the type of $e_R$ which is the Relationship meta model. The src and dst represent the connected source and target vertices.
- The attach relation between Point instances and Object instances: $e_A$= *(eatype, src, dst)*, $e_A \in E_A$. The eatype is the identifier of $e_A$. The src refers to the connected Point instance, the dst refers to the connected Object instance.

- The bind relation between Role instances and Object instances or Point instances: $e_B$= *(ebtype, src, dst)*, $e_B \in E_B$. The ebtype is the identifier of the bind relation. The src refers to the connected Object instance or Point instance, and the dst refers to the connected Role instance.

Base on the formal definition of KARMA model by graph theory, we can use the graph matching approach to realize the MP-based query of the KARMA model. In graph theory, graph pattern matching is based on the isomorphism of subgraphs (Larrosa et al., 2002) which requires finding an image (i.e. match) of a given graph (i.e. pattern graph) in another graph (i.e. source graph). Graph pattern matching can be divided into structural matching and semantic matching (Gallagher Brian, 2006). Structural matching is strictly based on structure similarity to find matches. However, semantic matching takes into account structural similarity as well as the types of vertices and edges and the attributes they owned. In the MP-based model query of the KARMA model, all elements have a type, and we can add constraints for the value of Property instances owned by elements that need to be queried. Therefore, we adopt graph pattern matching which not only considers structural similarity but also semantics constraints to realize MP-based model query. The formal definition of the MP based on graph theory is as in (3).

$$G_P = (V_{MP}, E_{MP}, constraintExpr?) \quad (3)$$

$G_P$ refers to the graph pattern, which is a specific subgraph of KARMA source model that need to be queried. The $V_{MP}$ is a finite set of $V_O$, $V_P$ and $V_R$, and the $E_{MP}$ is a finite set of $E_R$, $E_A$ and $E_B$. The $V_P$ and $E_P$ are used to describe the MPs of MP. The constraintExpr refers to semantic constraints imposed on $V_P$ and $E_P$, which are used to describe the $MP_C$ of MP.

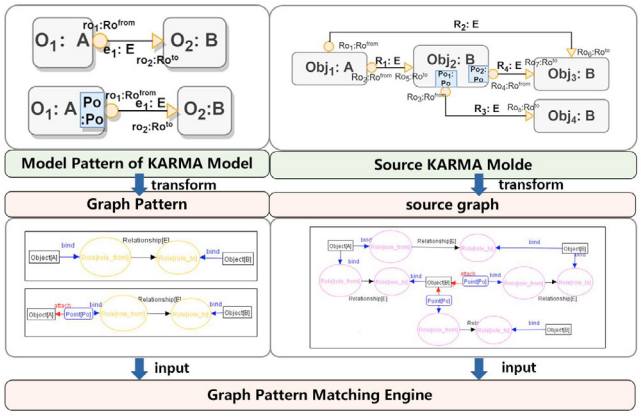


Figure 4 Model Query of KARMA Models Based on Graph Pattern Matching

Then we can transform the MP-based model query problem to a graph pattern matching problem. As shown in Figure 4, the model pattern is transformed to a graph pattern and the source KARMA model is transformed into the graph. Then the graph pattern and source graph are inputted into a graph pattern matching engine (GPME). The GPME can obtain all the subgraphs in the source graph that conform to the graph pattern, and turn them to the queried results of KARMA model to support the next step of code generation.

### *3.2 Design and implementation of code generation language*

The graph pattern matching approach provides a solution for MP-based query of KARMA models. However, the realization of this approach requires a medium to describe the model pattern and the source model to be queried. Therefore, this section focuses on the implementation of code generation language to support the description of model query and transformation.

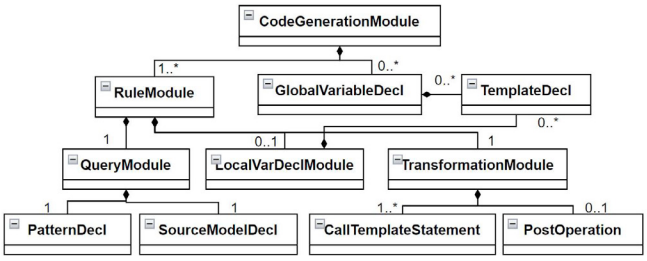


Figure 5 The Abstract Concepts for Code Generation of KARMA

The rule-based syntax to support code generation based on graph pattern matching is extended to KARMA. The extended abstract concepts for code generation are shown in Figure 5, The *CodeGenerationModule* is used to declare a code generation unit for KARMA models. The *GlobalVariableDecl* and the *LocalVarDeclModule* are used to declare global and local variables respectively. Variables of data types such as Int, Real, String, Boolean and File are supported. The focus of this paper is the *RuleModule* which is used to define code generation rules identified by the keyword ***rule***. The concrete syntax of *RuleModule* is shown in Figure 6. The *QueryModule* is used to describe the model query part of code generation. It is divided into two parts: *SourceModelDecl* and *PatternDecl*. The *SourceModelDecl* is used to describe source models that need to be transformed which is identified by the keyword ***in***.

```
rule ruleName {
    from//PatternDecl
        [modelElementDecl]+
        [binaryRelationStatement]+      ➡declare model pattern structure
        [cardinalityStatement]?
        [constraintStatement]? // declare model pattern constraints
    in modelType identifier// SourceModelDecl
    to // CallTemplateStatement
        String varIdentifier(
            varIdentifier <- tempalteName(args);
        )
     do { // PostOperation
         e.append(varIdentifier);
     }
}
```

Figure 6 The concrete syntax of RuleModule

The *PatternDecl* can define the MP which starts from the keyword ***from***. The *modelElementDecl* is used to declare OPPR instances existed in MP. The *binaryRelationStatement* is used to describe the relation between OPPR instances, for example, the “include” relation between the Object instance and the Point instance. The *cardinalityStatement* is used to define the cardinality of element which describes the number of occurrences of certain OPPR instances in the MP. The *constraintStatement* is used to describe the $MP_C$ in MP which consists of Boolean expressions.

The *TransformationModule* describes the transformation part and it is divided into two parts: *CallTemplateStatement* and *PostOperation*. The *CallTemplateStatement* is used to generate the target code from the information obtained in the model query part by calling the user-defined templates. The *PostOperation* is used to describe subsequent operations on the generated code, such as adding the code to a file.

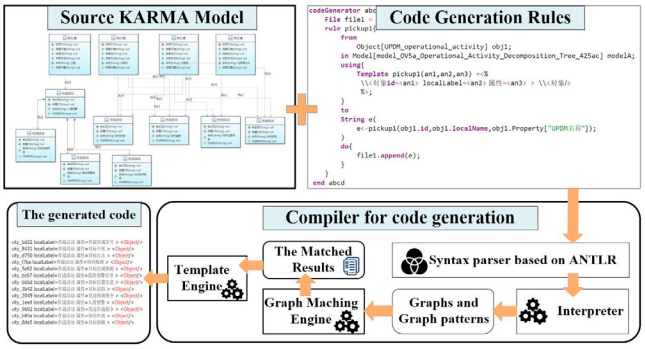


Figure 7 The execution process of code generation

Then, a compiler for code generation is developed. As shown in Figure 7, The compiler consists of a syntax parser, an interpreter, a GPME and a template engine. The syntax parser is developed based on ANTLR (Parr, 2013) to analyze code generation rules and generate abstract syntax trees. The interpreter extracts the information of model pattern and source model from the tree and transforms them to graphs and graph patterns, which is inputted into the GPME. We adopt the open-source tool AGG (Taentzer, 2003) as the GPME. After the execution of AGG, the matched results are inputted to the template engine, which is used to generate the target code by user-defined templates. Finally, the generated code can return to the MetaGraph workbench and the file specified by users.

## 4.CASE STUDY

In this section, a case of the unmanned vehicle system (UVS) is used to validate the availability of the proposed approach and language. The video of this case is available in YouTube (https://www.youtube.com/watch?v=J5rwb5WIWwQ&feature=youtu.be). The model is available in Gitee (https://gitee.com/zkhoneycomb/open-share/tree/master/code%20generation).

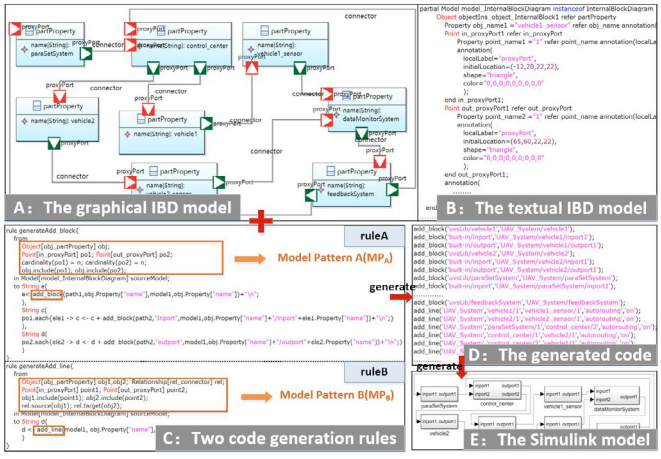


Figure 8 Generate .m file from IBD Model of UVS

The UVS consists of two unmanned vehicles, data monitoring subsystem, sensors and control center subsystem, etc. As shown in Figure 8-A, the graphical internal block diagram (IBD) is constructed by KARMA language to describe the internal components and their interaction of the UVS. The synchronized textual IBD model is shown in Figure 8-B. As shown in Table 1, different GOPPRR meta models are used to express different concepts in IBD. For example, the concept of partProperty can be described by an Object meta model, which can be used as the type of Object instances and expressed as "Object[partProperty]" in KARMA.

Table 1 Elements of KARMA to express concepts of IBD

| Meta models | IBD Concepts | KARMA Expression |
|---|---|---|
| Graph meta model | IBD | Graph[IBDdiagram] |
| Object meta model | partProperty | Object[partProperty] |
| Point meta model | in and out proxy port | Point[in_proxyPort], Point[out_proxyPort] |
| Relationship meta model | connector | Relationship[connector] |
| Role meta model | two ends of the connector | Role[connector_from], Role[connector_to] |

This case aims to transform the IBD model into a Simulink model by code generation to realize the simulation of UVS model. We adopt the method proposed by Sindico et al. (2011) to generate Simulink models. First, the mapping relations between IBD model and Simulink model is established in Table 2. Then the IBD model is transformed to the MATLAB model generation script through code generation. The model generation script consists of two main commands: *add_block* and *add_line*. The partProperty and proxy ports can be transformed to subsystem blocks and input or output blocks by generating the *add_block* command, and connectors between proxy ports can be transformed into lines between input and output blocks by generating the *add_line* command.

Table 2 The mapping relations between IBD and Simulink

| Elements in IBD | Elements in Simulink |
|---|---|
| partProperty | subsystem block |
| proxy port and full port | input/output block |
| connector | line |

Based on the mapping relations, two code generation rules are defined to realize the generation of the Simulink model. As shown in Figure 8-C, the model pattern A ($MP_A$) in ruleA declares a model structure consisting of an Object instance of type "Object[partProperty]" and Point instances of type "Point[in_proxyPort]" or "Point[out_proxyPort]" it owns. The obtained information by the query of $MP_A$ is inputted into the "add_block" template. In addition, the model pattern B ($MP_B$) in ruleB describes a model structure consisting of two Object instances (obj1 and obj2) of type "Object[partProperty]", a Relationship instance of type "Relationship[connector]", a Point instance of type "Point[in_proxyPort]" owned by obj1 and a Point instance of type "Point[out_proxyPort]" owned by obj2. The obtained information by the query of $MP_B$ is inputted into the "add_line" template. Then, two rules are compiled by

the compiler for code generation. The $MP_A$ and $MP_B$ are transformed to two graph patterns, and the IBD model is transformed into a graph. After the execution of the AGG of compiler, as shown in Figure 8-D, a MATLAB script with *add_block* and *add_line* commands is generated by inputting the queried information into the template engine. Finally, the script is executed by MATLAB and the Simulink model is loaded as shown in Figure 8-E.

From the case study, the availability of the proposed approach and KARMA language is demonstrated. With the help of the KARMA, the IBD model is expressed in formalized text. The code generation of more complex model structure with several model elements are realized by the extended KARMA.

## 5. CONCLUSIONS AND FUTURE WORKS

In the paper, we proposed a unified approach for the formalism and code generation of multi-domain system models. The KARMA language based on GOPPRR-E meta-modeling approach was used to uniformly express the heterogeneous system models. Then we proposed a code generation approach based on graph pattern matching which can support the MP-based model query of KARMA models. The KARMA language is extended to support the code generation based on the proposed approach. Finally, through a case of transforming the IBD model of UVS into a MATLAB file, the availability of the proposed approach and language was validated.

In the future, the boundaries of the model structure that the model pattern can describe will be defined in detail. Then, a validity-checking module for model pattern will be added to the code generation compiler to check whether the user-defined MP conforms to the defined boundaries. Moreover, the correctness checking mechanism of the generated code will be considered, which is used to check the syntax and semantics of the generated code.

## ACKNOWLEDGEMENT

This work was supported by The National Key Research and Development Program of China (No. 2020YFB1708100) and the CN pre-study common technology (No. 50923010101).

## REFERENCES

Champin, PA., Solnon, C. (2003). Measuring the Similarity of Labeled Graphs. In Case-Based Reasoning Research and Development. ICCBR 2003. Lecture Notes in Computer Science(), vol 2689

Gallagher Brian. (2006). Matching structure and semantics: A survey on graph-based pattern matching. AAAI Fall Symposium - Technical Report, FS-06-02, 45–53.

Grunske, L., Geiger, L., Zündorf, A., Van Eetvelde, N., Van Gorp, P., & Varro, D. (2005). Using graph transformation for practical model-driven software engineering. In Model-driven Software Development (pp. 91-117). Springer, Berlin, Heidelberg.

Hart, L. E. (2015). Introduction to model-based system engineering (MBSE) and SysML. In Delaware Valley INCOSE Chapter Meeting. Vol. 30.

Kelly, S., & Tolvanen, J. P. (2008). Domain-specific modeling: enabling full code generation. John Wiley & Sons.

Kern, H., Hummel, A., & Kühne, S. (2011). Towards a comparative analysis of meta-metamodels. In Proceedings of the compilation of the co-located workshops on DSM'11 (pp. 7-12).

Lu, J., Wang, G., Ma, J., Kiritsis, D., Zhang, H., & Törngren, M. (2020). General Modeling Language to Support Model‐based Systems Engineering Formalisms (Part 1). INCOSE International Symposium, 30(1), 323–338.

Larrosa, J. and Valiente, G. (2002). Constraint satisfaction algorithms for graph pattern matching. Mathematical Structures in Computer Science, 12(4):403–422.

Magic, N. (2016). Magic Draw. https://www.nomagic.com /products/magicdraw.html.

OMG, Object Management Group. (2017). Unified Modeling Language: Superstructure. Version 2.5.1.

OMG. (2011). Business Process Model and Notation. Version 2.0.

OMG. (2017). System Modeling Language. Version 1.5.

OMG. (2000). Meta Object Facility Specification.

Parada, Abilio & Siegert, Eliane & Brisolara, Lisane. (2022). GenCode: A tool for generation of Java code from UML class models.

Parr, T. (2013). The definitive ANTLR 4 reference.

Sebastián, G., Gallud, J. A., & Tesoriero, R. (2020). Code generation using model driven architecture: A systematic mapping study. Journal of Computer Languages, 56, 100935.

Sindico, A., Di Natale, M., & Panci, G. (2011). Integrating SysML with Simulink using Open-source Model Transformations. In SIMULTECH (pp. 45-56).

Steinberg, D., Budinsky, F., Merks, E., & Paternostro, M. (2008). EMF: eclipse modeling framework. Pearson Education.

The Eclipse Foundation. (2021). Eclipse's Model To Text (M2T) component. https://www.eclipse.org/modeling/ m2t/?project=jet.

Taentzer Gabriele. (2003). AGG: A graph transformation environment for modeling and validation of software. In International workshop on applications of graph transformations with industrial relevance (pp. 446-453).

Usman, M., & Nadeem, A. (2009). Automatic generation of Java code from UML diagrams using UJECTOR. International Journal of Software Engineering and Its Applications, 3(2), 21-37.