2020 5th International Conference on Mechanical, Control and Computer Engineering (ICMCCE) | 978-1-6654-2314-4/20/$31.00 ©2020 IEEE | DOI: 10.1109/ICMCCE51767.2020.00266

A Model Query Language for Domain-Specific Models

Jiangmin Guo
Beijing Institute of Technology
Beijing, China
guojmbucter@163.com

Jie Ding
Beijing Institute of Technology
Beijing, China
15901205268@163.com

Jinzhi Lu*
EPFL - Ecole Polytechnique Fédérale de Lausanne
Lausanne, Switzerland
jinzhi.lu@epfl.ch

Guoxin Wang
Beijing Institute of Technology
Beijing, China
wangguoxin@bit.edu.cn

***Abstract*—Model queries play a crucial role in the Model-driven development processes, particularly for Domain-Specific Modeling (DSM) and Model-based Systems Engineering (MBSE). The model queries are also regarded as the cornerstone for model-driven development activities, such as code generation, model transformation, and model constraints checking. The GOPPRR metamodeling approach is widely used to formalize the domain-specific models. Based on this approach, the KARMA language has been proposed to formalize models, metamodels, and code generation but lacks support for the model querying. This paper proposed one query language based on the GOPPRR metamodeling approach extended from the KARMA language to realize the unified query formalisms for multi-domain models. Finally, a case in a vehicle tracking system development is used to verify the availability of model query language, which is implemented in a domain modeling tool, *MetaGraph*.**

Keywords-Domain-Specific Language; model query language; model-driven development; Model-based Systems Engineering

I. INTRODUCTION

Models are primary artifacts in the development process of Model-Driven Engineering (MDE), particularly for Domain-Specific Modeling (DSM) and Mode-based Systems Engineering (MBSE), which are expressed by general and Domain-Specific Languages (DSL) [1]. Model queries are cornerstones of the MDE process and are widely used in model development and analysis, such as model transformation, code generation, and model checking. Model queries provide a means to capture model components and information according to the specifications [2], which are formalized by the model query languages, such as EMF-INCQUERY, EMF Model Query [3] and OCL [4]. Moreover, the query enables the realization of design automation to develop complex systems by model transformation[5].

In the current industrial development process of complex systems, multiple domain models are involved, which are formalized by the corresponding DSL. A metamodeling approach, the GOPPRR approach, is proposed to formalize the DSL[6]. The GOPPRR approach using *Graph-Object-Property-Port-Role-Relationship* meta-meta models to formalize the functional graphical DSL [7]. Based on this approach, the KARMA language has been developed in paper [8] to support the formalize models, metamodels, code generation, and model transformation during the whole lifecycle of systems in MBSE. However, the KARMA language lacks the support for model querying for GOPPRR models.

In this paper, one textual query language is proposed, which is developed based on the model pattern mechanisms for querying domain-specific models based on the GOPPRR approach. This query language supports the complex model queries by synthesizing model patterns and is extended from KARMA language. In detail, the KARMA language first formalizes the domain-specific model based on the GOPPRR approach. Then the query language describes the model query specification using model patterns. Finally, a part of the model elements is captured according to the defined model patterns. Therefore, the query language implements the unified queries specification for the domain-specific models based on the KARMA language.

The rest of the paper is organized as follows. Section 2 describes the query language for GOPPRR models based on three types of basic model patterns. Then, a case is introduced using one modeling tool *MetaGraph* [1] to demonstrate the availability of the query language in Section 3. Section 4 concludes the paper with future research work.

II. A QUERY LANGUAGE FOR DOMAIN-SPECIFIC MODELS

In this section, a query language for Domain-Specific models built on the GOPPRR metamodeling approach is proposed to formalize the query process when developing a complex system. Firstly, the basic of the GOPPRR metamodeling approach and the KARMA language is introduced. Then three essential model patterns that specify how the components of the model connected each other are summarized in the GOPPRR model. Finally, based on the model pattern, we demonstrate the syntax of the query language, which supports the queries of GOPPRR models concisely and intuitively.

[1] MetaGraph is a multi-architecture modeling tool developed by Z.K. fC http://www.zkhoneycomb.com/.

A. *GOPPRR Metamodeling Approach*

In the development process of complex systems, multiple domains are involved, where domain-specific models are used to formalize the domain concepts. Generally, the domain-specific models are described by the DSLs based on an M3-M0 modeling framework [9] shown in Table 1. The GOPPRR approach uses *Graph-Object-Property-Port-Role-Relationship* as meta-meta models to define the DSL[10], and which is considered expressive among a huge number of metamodeling languages.

TABLE I. M3-M0 MODENLING FRAMEWORK

Level	Description
M3 (Meta-meta-model)	The model used to specify modeling languages, which are six GOPPRR elements in our approach.
M2 (Meta-model)	The model used to specify domain models, which are instances of meta-meta-model.
M1 (model)	Models are used to describe system artifacts in the development of the complex system.
M0(System view)	System view in the real world.

TABLE II. THE BINARY RELATIONSHIP BETWEEN META-META MODELS

	Graph	***Object***	***Relationship***	***Role***	***Point***
Graph	-	containIn	containIn	-	-
Object	include	connectWith	-	bindWith	containIn
Relationsh ip	include	-	-	isStartOf isEndOf	-
Role	-	bindWith	startFrom endTo	-	bindWith
Point	-	include	-	bindWith	-
Property	have	have	have	have	have

Figure 1 shows the simplified concepts of the GOPPRR modeling language in the form of the UML class diagram, and the complete meta-meta model abstract syntax can be found in the paper [6]. Except for six key meta-meta models, the binary relationships[11] are defined to constrain the connection rule between the meta-meta models, as shown in table 2, such as the *Object* can be contained in the *Graph*.

- Graph refers to a set of Objects and Relationships, which is represented as one diagram in a window (similar to an integrated concept of a class diagram and package in UML). The Graph can be instantiated as a graph metamodel denoted by *Graph[type]* according to the domain concept[12].

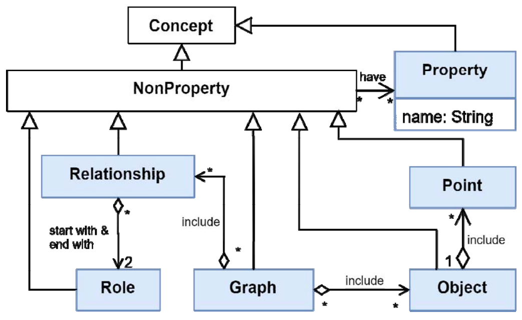


Figure 1. Simplified concepts of the GOPPRR metamodeling language

- Object represents one entity in the Graph (similar to the concept of class in UML). The Object can be instantiated as an object metamodel of domain-specific entity denoted by *Obejct[type]*. It has an optional set of Points.
- Property refers to the attribute specified the value and value's datatype denoted by *Porperty[type]*, which can be held by other five meta-meta models.
- Point is an optional element, which is contained by one object instance denoted by *Point[type]*. Generally, objects can be bind with role directly, but with point it can be additionally specified, through which additional semantics can be expressed.
- Relationship refers to a kind of connection between the different two or more Points and Objects, denoted by *Relationship[type]*.
- Role is used to define the connection rules to target elements (Object or Point) in a specific relationship, binding with Object and Point directly denoted by *Role[type]*.

Based on the GOPPRR approach, a textual modeling language, KARMA [8], has been developed to formalize the specific-domain metamodels and models. In detail, the domain-specific metamodel is defined based on the GOPPRR meta-meta model. Then based on the metamodel, the domain-specif models are instantiated to build models. In the above process, all the metamodels and models are described by KARMA language uniformly saved as structural text in the .kar file. Besides, the KARMA language supports formalisms for code generation and model transformations.

B. *Model Query Process*

The model query refers to the execution to captures model elements (such as *Object* and *Relationship* instances) on a GOPPRR model *M*, which consists of many model parts (such as *part1*, *part2* and *part3* shown in Figure 2). These parts can be divided into three types according to the connection between model elements specified by the *model pattern*. The *model pattern* is a structure that consists of model elements and specifies how the elements are connected to each other. Three kinds of model patterns are summarized (simple pattern, cardinality pattern and grouping pattern) introduced in the next section.

As shown in Figure 2, when we query a GOPPRR model *M*, the query language is used to describe the query specification by defining model patterns based on the queried model. Multiple model patterns can be defined in the model query language, and if a part of the model is matched by model pattern *P*, the set of the matched elements will be captured. We explain that the model *part* is *matched* by model pattern *P* if all the connection constraints on elements in the model are satisfied with the constraints defined in the pattern *P*. The captured elements are formalized as $\{(T_1, E_1),(T_2, E_2),\ldots\}$, which is the set of the maps between an element (e.g., T_1, T_2) in pattern and corresponding element (e.g., E_1, E_2) in the queried model.

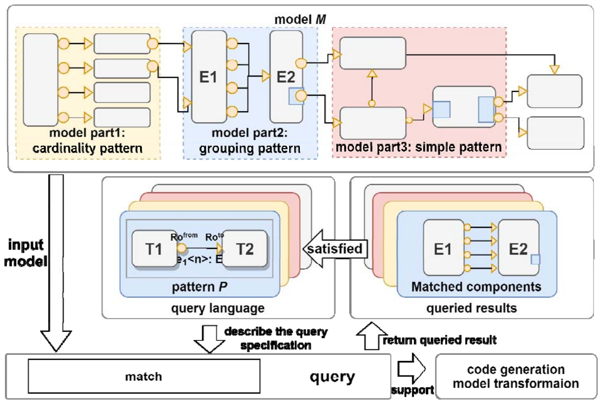

model M
model part1: cardinality pattern
model part2: grouping pattern
model part3: simple pattern
input model
pattern P
query language
satisfied
Matched components
queried results
describe the query specification
return queried result
match
query
support
code generation model transformaion

Figure 2. The model query process

C. *The Model Pattern for GOPPRR model*

1) Simple pattern

The simple pattern refers to a specification that every queried GOPPRR element in the model is enumerated in the pattern, and the connection between them is specified following a binary relationship. As shown in Figure 3-A and B, two types of simple patterns are summarized in the GOPPRR model.

- Directly binding pattern refers that one object instance O_1 (with metamodel A) connect with another object instance (with metamodel B), where all elements are enumerated. In detail, O_1 and O_2 bind with role instance ro_1 (with metamodel Ro^{from}) and ro_2 (with metamodel Ro^{to}) respectively, and both ro_1 and ro_2 are included in the relationshiop instance e_1 (with metamodel E).
- Indirectly binding pattern refers to the connection between two object instance O_1, O_2, the object instance O_1 bind with ro_1 through a point instance Po_1 (with metamodel Po) other than directly binding. Similarly, the O_2 binds with ro_2 through Po_2. There is at least one point on the connection between O_1 and O_2.

In the model (shown in Figure 3-C), two valid model parts can be found using the directly binding pattern (shown in Figure 3-A), which are the connection from Obj_1 to Obj_2 and the connection from Obj_1 to Obj_3 denoted by {(O_1, Obj_1), (ro_1, Ro_2), (e_1, R_1), (ro_2, Ro_5), (O_2, Obj_2) } and {(O_1, Obj_1), (ro_1, Ro_1), (e_1, R_2), (ro_2, Ro_6), (O_2, Obj_3)}, where the object instances bind with role instances directly. Similarly, two valid model parts {(O_1, Obj_2), (Po, Po_2), (ro_1, Ro_4) (e_1, R_4), (ro_2, Ro_7), (O_2, Obj_3)} and {(O_1, Obj_2), (Po, Po_1), (ro_1, Ro_3), (e_1, R_3), (ro_2, Ro_8), (O_2, Obj_4)} are matched for the indirectly binding pattern shown in Figure 3-B, where object instances bind with role instances through point instance.

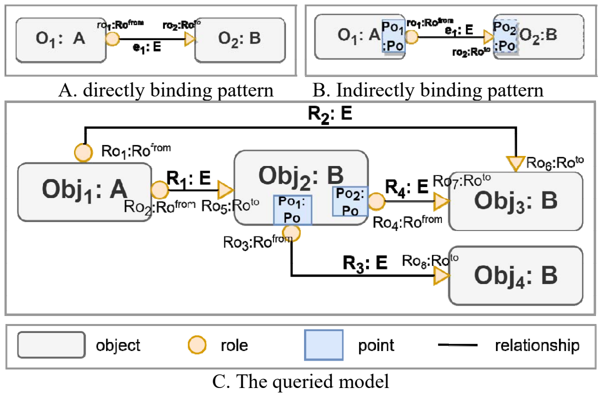


Figure 3. Simple pattern specfication

2) Cardinality pattern

When querying the more complex model, we can not enumerate every GOPPRR element in a simple pattern. For example, if the queried model contains n object instance, it is impossible to specify them using the *simple pattern* in an enumerated manner. So the *cardinality pattern* is defined to describe that a GOPPRR element in the pattern can appear n times in the model if the cardinality is specified to n. Three types of cardinality pattern for *Object, Relationship,* and *Point* are introduced, as shown in Figure 4.

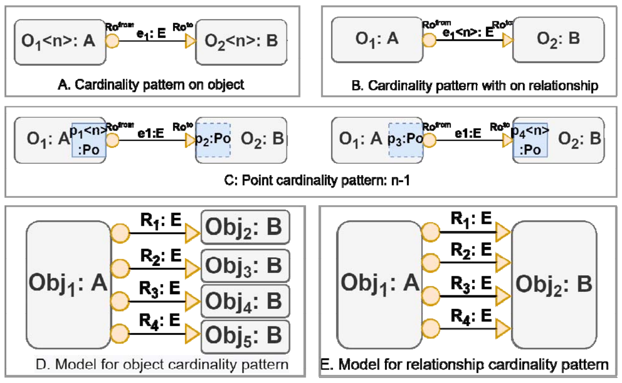


Figure 4. Specification for the pattern with cardinality

The cardinality describes the number of certain element occurrences in the model, which is specified with either a fixed number or a range (x, y) from x to y. Moreover, when defining the cardinality pattern, the wildcard '*', '+' and '?' can be used to represents cardinality is "no limit", "at least one" and "optional".

- The object cardinality pattern describes the occurrence of an object instance in the model. As shown in Figure 4-A, the pattern specifies that n O_1 object instance (with metamodel A) and n O_2 object instance (with metamodel B) can be found in the model. Besides, the connection between each pair of object instances must conform to the connection defined in the pattern. One model part (shown in Figure 4-D) can be found if the cardinality pattern (Figure 4-A) is specified to 1 and 4. The match is denoted as {(O_1, Obj_1), (O_2, {Obj_2, Obj_3, Obj$_4$, Obj$_5$})}, in which O_1 in occurrence one and O_2 occurrence four.

- The relationship cardinality pattern describes the occurrence of the relationship instance. One model part (shown in Figure 4-E) can be found if the relationship cardinality pattern (Figure 4-B) is specified to four. The queries result is denoted as {(O$_1$, Obj$_1$), (O$_2$, Obj$_2$), (e$_1$, {R$_1$, R$_2$, R$_3$, R$_4$})} where the role is omitted for brevity.
- The n-1 point cardinality pattern describes the occurrence of the point instance, where all the point instances are contained in two object instances. The n-1 point cardinality describes that n point in one object instance can connect to (or be connected to) one another element (object instance or point instance). As shown in Figure 4-C, *n* point p$_1$ (with metamodel *Po*) connect to the object O_2 (with metamodel *B*) through a certain role and relationship. One model part can be found in the model shown in (shown in Figure 5-D), which are {(p$_1$, {p4, p5}), (O$_2$, Obj$_4$)} where role and relationship bindings are omitted for brevity.

3) Grouping pattern

In order to specify the more complex model querying, the *grouping pattern* is proposed to describe that using multiple patterns (called sub-pattern) to describe the GOPPRR model elements. Another conception in grouping pattern is the host pattern, which represents the set of the sub-patterns. In the sub-patterns, either the simple pattern or cardinality pattern can be used. We can also describe the cardinality of the sub-pattern, which means that the sub-pattern must occurs *n* times in the host pattern when cardinality *n* is specified. Figure 5-A shows a host pattern consists of two sub-pattern *pattern1* and *pattern2*, and *pattern1* belongs to a simple pattern while *pattern2* is a cardinality pattern. The sub-pattern *pattern1* specifies the connection between Po_1 and *ro*, marked as P(Po$_1$, ro). The sub-pattern *pattern2* belongs to the cardinality pattern, which specifies that object O_2 (with metamodel *B*) contains *n* points (with metamodel *Po*), and the *pattern2* is marked as P(Po$_2$<n>, O$_2$).

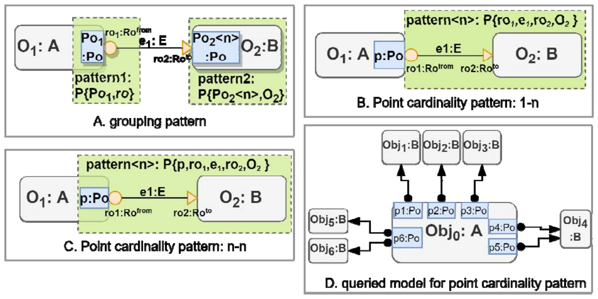


Figure 5. Grouping pattern and point cardinality pattern

- The 1-n point cardinality pattern shown in Figure 5-B expresses that a point instance (with metamodel *Po*) connects to *n* object instance (with metamodel *B*) through a certain role and relationship. Figure 5-B represents one host pattern, and n sub-pattern is contained by specifying cardinality on sub-pattern. Moreover, the point instance *p* is defined in the host pattern, and the object instance O_2 is defined in the sub-pattern pattern. One model part will be matched when the pattern used to the model (Figure 5-D), which is denoted as {(p, p6), (O$_2$, {Obj$_5$, Obj$_6$})}.
- The n-n point cardinality pattern (Figure 5-C) expresses that the object instance O_1 ontains *n* point instances (with metamodel *Po*). And for each point *p*, there is a corresponding object instance O_2 connected to it. One sub-pattern pattern describes such point object pairs, which occurrence *n* times by defining cardinality. One model part (Figure 5-D) is matched, which is {(O$_1$, Obj$_0$), (pattern, {(p, p1), (O$_2$, Obj$_1$)}, {(p, p2), (O$_2$, Obj$_2$)}, {(p, p3), (O$_2$, Obj$_3$)})}.

D. *A Query Langauge for GOPPRR Model*

This section demonstrates a language for querying domain-specific models conforming to GOPPRR metamodels, which achieve a unified description for the cross-domain model query. According to the GOPPRR metamodeling language syntax, the domain metamodels can be classified into six GOPPRR categories denote by *Graph[type]*, *Object[type]*, *Point[type]*, *Relationship[type]*, *Role[type]* and *Property[type],* where *type* identify the type of the metamodel. Therefore, based on the six types of metamodels, the query language can uniformly formalize model queries in different domains. Based on the basic pattern mentioned above, the syntax of the language is demonstrated to specify complex structural model queries concisely and expressively. Moreover, the attributes of the model elements can be constrained based on the value of the component's property.

1) Pattern concrete syntax

The pattern concrete syntax consists of *pattern name*, *pattern parameters*, and *pattern body*. The pattern is distinguished by *pattern name*, and each pattern can define multiple parameters with metamodel to declare the elements in the pattern. The connections between elements are specified in the *pattern body* in forms of binary relationship according to table 2.

2) Simple pattern syntax

The syntax in Figure 6 shows how to describe the *simple pattern* shown in Figure 3-A. The pattern *simpleMatching* declares the connection among parameters *o1*, *ro1*, *ro2*, *e1* and *a2*. In detail, Line 3 captures the semantic that one instance *o1 (*with metamodel *Object[A]*) binds with *ro1* (with metamodel *Role[RO_from])* directly, and Line 4 and 5 describe the relationship *e1* start form *ro1* and end to *ro2*. The syntax for the binary relationship between metamodels are denoted by "*.binary_name()*", and the *have* relationship between property and other five types of metamodels are denoted by ". *Property_name*".

3) Cardinality pattern syntax

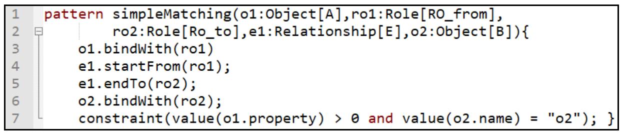
Moreover, the cardinality pattern is implemented by adding cardinality restrictions to matching parameters or sub-patterns, which are usually used combined with the *constraint* syntax and denoted as *cardinality()*. As shown in Figure 7, the cardinality of the sub-pattern (see Line 7) is specified to 4, and the cardinality of the instance of the object *o1* (see Line 6) is specified to 1.

4) Grouping pattern syntax

The concept of *grouping pattern* is supported by the calling syntax denoted by the *match* keyword. The host pattern specifies the connections of the elements by declaring the sub-pattern. An example (shown in Figure 7) shows that elements denoted by parameters (*p, o2, ro1, ro2, e1*) is matched by a sub-pattern called *subPatternPointToObject.* And the host pattern *pointCardinalityNTon* declares one sub-pattern *subPatternPointToObject*, where the *Point[Po]* instance *p* is specified must be included in o_1 shown as Line 4 (Figure 7).

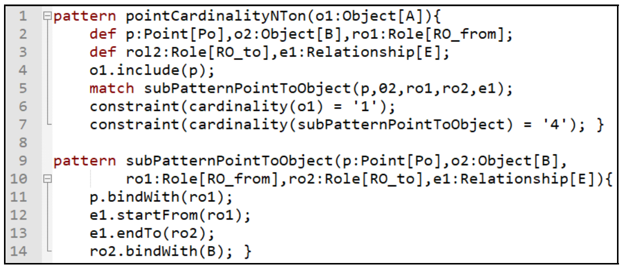
5) Constraints syntax

The *constraints* syntax filter the queried elements by defining Boolean expressions. Such syntax can filter the elements by checking attributes of elements based on the property value. See Figure 6, Line 7, the object instance *o1* and *o2* are filtered by constraining the property value.



```
1  pattern simpleMatching(o1:Object[A],ro1:Role[RO_from],
2          ro2:Role[Ro_to],e1:Relationship[E],o2:Object[B]){
3      o1.bindWith(ro1)
4      e1.startFrom(ro1);
5      e1.endTo(ro2);
6      o2.bindWith(ro2);
7      constraint(value(o1.property) > 0 and value(o2.name) = "o2"); }
```

Figure 6. The language syntax for the simple pattern



```
1  pattern pointCardinalityNTon(o1:Object[A]){
2      def p:Point[Po],o2:Object[B],ro1:Role[RO_from];
3      def rol2:Role[RO_to],e1:Relationship[E];
4      o1.include(p);
5      match subPatternPointToObject(p,02,ro1,ro2,e1);
6      constraint(cardinality(o1) = '1');
7      constraint(cardinality(subPatternPointToObject) = '4'); }
8
9  pattern subPatternPointToObject(p:Point[Po],o2:Object[B],
10         ro1:Role[RO_from],ro2:Role[RO_to],e1:Relationship[E]){
11     p.bindWith(ro1);
12     e1.startFrom(ro1);
13     e1.endTo(ro2);
14     ro2.bindWith(B); }
```

Figure 7. The language syntax for grouping pattern

III. Case Study

In this paper, a case of the vehicle tracking system is used to verify the usability of the query language. One architecture model is used to formalize and analyze the tracking system using one customized DSL based on the GOPPRR metamodeling approach. Figure 8 shows the metamodel of the tracking system architecture. Based on the metamodel, the architecture model of the tracking system is built, shown in Figure 9-A.

An essential role of the architecture models is to simulate the runtime states by transforming the model to the Simulink model with code generation. Firstly, the architecture model (Figure 9-A)is built based on the DSL and formalized by the KARMA language. Then the query language (Figure 9-C) is used to capture the required model components described by the KARMA segment (Figure 9-B) in the architecture model. Finally, based on the queried components, the code generation supported by KARMA is used to generate the .m code (Figure 9-D), and the Simulink model (Figure 9-E) is built by parsing the code.

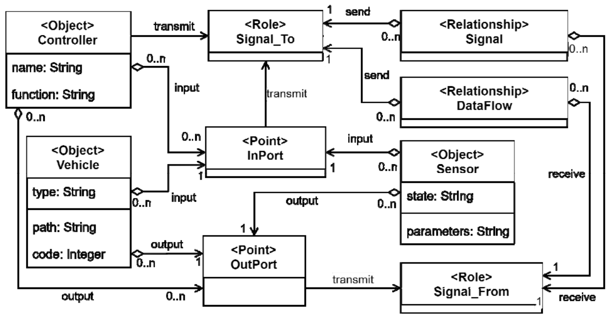


Figure 8. UML class diagram of vehicle tracking system metamodel

1) Query dispatch-center components.

The dispatch-center is an object instance of Controller, which is connected to multiple Vehicle instances through a port (with type *Point[OutPort]*). It can be queried using pattern *schedulingVehicle* based on the n-n point cardinality pattern. The sub-pattern *connectVehicle* is declared to constraints the connection of the *Point[Output]* instance contained in the *Object[Controller]* instance. The details of the pattern are shown in Figure 10-B, and the KARMA language formalisms of queried model elements are shown in Figure 10-A.

2) Query vehicle components.

The pattern *transVehicle* is used to query elements by checking the cardinality of the connected sensors, which conforms to the 1-n point cardinality pattern.

3) Query the sensors and feedback unit.

The *position sensor* and the *distance sensor* connect with the *feedback unit* through one input point, which conforms to the n-1 point cardinality pattern. The pattern *sensorAndFeedBack* is used to capture these elements.

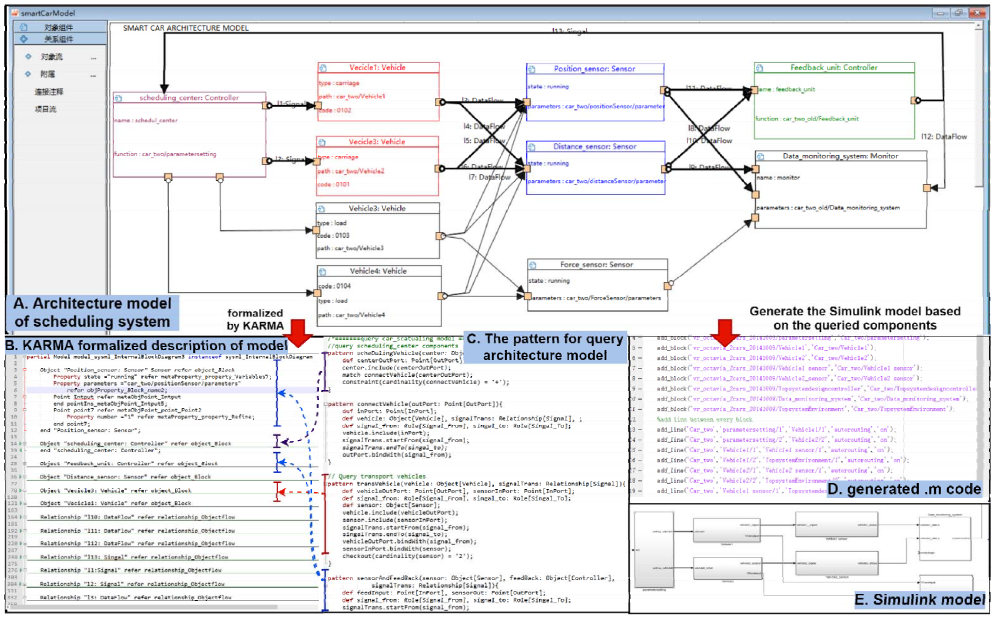

smartCarModel
SMART CAR ARCHITECTURE MODEL
A. Architecture model of scheduling system
formalized by KARMA
B. KARMA formalized description of model
C. The pattern for query architecture model
Generate the Simulink model based on the queried components
D. generated .m code
E. Simulink model
scheduling_center: Controller
Vecicle1: Vehicle
Vecicle3: Vehicle
Vehicle3: Vehicle
Vehicle4: Vehicle
Position_sensor: Sensor
Distance_sensor: Sensor
Force_sensor: Sensor
Feedback_unit: Controller
Data_monitoring_system: Monitor

Figure 9. Generate a Simulink model based on the query language in the tracking system case

```
Object "scheduling_center: Controller" refer object_Block
    Property state ="schedul_center" refer objProperty_Block_name2;
    Property function ="car_two/parametersetting"
        refer metaProperty_property_Variables7;
    Point "Output" refer metaObjPoint_point_Point2
    end "Output";
    Point "Output" refer metaObjPoint_point_Point2
    end "Output";
end "scheduling_center: Controller";
```

A. KARMA formalisms for queried dispatch-center components

```
pattern scheDulingVehicle(center: Object[Controller]){
    def centerOutPort: Point[OutPort];
    center.include(centerOutPort);
    match connectVehicle(centerOutPort);
    constraint(cardinality(connectVehicle) = '+');
}
```

B. query pattern for dispatch-center components

Figure 10. The query for dispath-center components

TABLE III. THE REFERRED PATTERNS IN THE CASE STUDY

Used patterns in the case study	Referred patterns
scheDulingVehicle	Simple pattern, grouping pattern, n-n point cardinality patter
connectVehicle	Simple pattern
transVehicle	Simple pattern, 1-n point cardinality pattern
sensorAndFeedBack	Simple pattern, n-1 point cardinality pattern

The queried components of the model are shown in the Figure 9-A, which are matched by the corresponding pattern shown in Figure 9-C marked with the same color. Moreover, the KARMA language based on the GOPPRR approach is used to formalize these model components (shown in Figure 9-B) and also realizes the code generation to generate the Simulink model (shown in Figure 9-D, Figure 9-E). On the tool level, *MetaGraph*, a multi-architecture modeling tool, provides a framework for realizing modeling, code generation, and model querying based on the KARMA language.

From the case study, the availability of the query language is demonstrated. Based on the defined domain metamodel, more complex model components are queried by the query language according to the basic pattern. With the KARMA language, the queried model components are formalized and transformed into Simulink models using code generation. Compared with the existing query language, this query language is based on the GOPPRR metamodeling approach and compatible with the KARMA language realizes the unified query of multi-domain models.

IV. Conclusion

In this paper, a query language for GOPPRR models is proposed, which is extended from the KARMA language. Firstly, the basic conception of the GOPPRR approach and the KARMA language is introduced. Then three types of model patterns are defined to support the query language formalisms for GOPPRR models. Moreover, the syntax of the query language is demonstrated to formalize the model pattern. Finally, based on the case study, we identify that the query language can realize the model query for domain-specific models.

As the future, a static type reference mechanism of query language will be developed in order to provide model context information for the convenience of developers.

References

[1] J. Sprinkle, B. Rumpe, H. Vangheluwe, and G. Karsai, “3 Metamodelling,” in Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), vol. 6100 LNCS, 2010, pp. 57–76.

[2] G. Bergmann, Z. Ujhelyi, I. Ráth, and D. Varró, “A Graph Query Language for EMF Models,” in Lecture Notes in Computer Science

(including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), vol. 6707 LNCS, 2011, pp. 167–182.

[3] Á. Hegedüs, Á. Horváth, I. Ráth, and D. Varró, "Query-driven soft interconnection of EMF models," 2012, doi: 10.1007/978-3-642-33666-9_10.

[4] T. Clark and J. Warmer, Object Modeling with the OCL, vol. 2263. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002.

[5] A. Agrawal, G. Karsai, and F. Shi, "Graph Transformations On Domain-Specific Models," Under Consid. Publ. J. Softw. Syst. Model., no. Mic, pp. 1–43, 2003, [Online]. Available: http://www.sciences.univ-nantes.fr/lina/atl/www/papers/ext/Agrawal_A_11_0_2003_Graph_Tran.pdf.

[6] H. Kern, A. Hummel, and S. Kühne, "Towards a comparative analysis of meta-metamodels," in Proceedings of the compilation of the co-located workshops on DSM'11, TMC'11, AGERE!'11, AOOPES'11, NEAT'11, & VMIL'11 - SPLASH '11 Workshops, 2011, vol. 1, p. 7, doi: 10.1145/2095050.2095053.

[7] V. Dimitrieski, M. Čeliković, V. Ivančević, and I. Luković, "A Comparison of Ecore and GOPPRR through an Information System Meta Modeling Approach," no. July, 2012.

[8] J. Lu, G. Wang, J. Ma, D. Kiritsis, H. Zhang, and M. Törngren, "General Modeling Language to Support Model-based Systems Engineering Formalisms (Part 1)," INCOSE Int. Symp., vol. 30, no. 1, pp. 323–338, Jul. 2020, doi: 10.1002/j.2334-5837.2020.00725.x.

[9] H. Vangheluwe, J. De Lara, and P. J. Mosterman, "An introduction to multiparadigm modelling and simulation," Ais2002, 2000.

[10] Metacase, "The Graphical Metamodeling Example," p. 20, 2018, [Online].

[11] E. Biermann, C. Ermel, and G. Taentzer, "Formal foundation of consistent EMF model transformations by algebraic graph transformation," Softw. Syst. Model., vol. 11, no. 2, pp. 227–250, May 2012, doi: 10.1007/s10270-011-0199-7.

[12] A. A. Ephremides, "A Foundation for Multi-Paradigm Modelling," vol. 1, 2001.