Proceedings of the of the 17th Conference on Computer Science and Intelligence Systems pp. 919–928
DOI: 10.15439/2022F129
ISSN 2300-5963 ACSIS, Vol. 30

Beyond Low-Code Development: Marrying Requirements Models and Knowledge Representations

Kamil Rybiński, Michał Śmiałek
Warsaw University of Technology, Poland
Email: {kamil.rybinski, michal.smialek}@pw.edu.pl

***Abstract*—Typical Low-Code Development platforms enable model-driven generation of web applications from high-level visual notations. They normally express the UI and the application logic, which allows generating the frontend and basic CRUD operations. However, more complex domain logic (data processing) operations still necessitate the use of traditional programming. This paper presents a visual language, called RSL-DL, to represent domain knowledge with complex domain rules aligned with requirements models. The language synthesises and extends approaches found in knowledge representation (ontologies) and software modelling language engineering. Its purpose is to enable a fully automatic generation of domain logic code by reasoning over and reusing domain knowledge. The language's abstract syntax is defined using a meta-model expressed in MOF. Its semantics is expressed with several translational rules that map RSL-DL models onto typical programming language constructs. The rules are explained informally in natural language and formalised using a graphical transformation notation. It is also supported by introducing an inference engine that enables processing queries to domain models and selecting appropriate invocations to generated code. The presented language was implemented by building a dedicated model editor and transformation engine. It was also initially validated through usability studies. Based on these results, we conclude that declarative knowledge representations can be successfully used to produce imperative back-end code with non-trivial logic.**

I. Introduction

THE TERM "Low-Code Software Development" (LCSD) has emerged as a new approach to application development where only a limited amount of coding is required. Since its emergence around seven years ago [25], the term was used in the industry to label cloud-based development platforms that use visual notations to reduce the need for traditional programming. Research on low-code approaches is currently yet sparse. However, just recently, it has been observed that LCSD can be seen as a subdomain [7] or as overlapping [27] with the Model-Driven Software Development (MDSD, MDD), where it concentrates on automatic generation of data-rich web/mobile applications from visual specifications (models).

Our research was done in the context of the ReDSeeDS[1] platform [33]. The system was created before the emergence of the low-code movement, but it certainly fulfils the definition of LCSD. It uses precisely specified requirements models: use cases, scenarios in constrained language, and visual domain vocabularies. Similarly to low-code platforms, these artefacts are represented with a visual language called RSL (Requirements Specification Language) [21]. Specifications expressed in RSL allow for the generation of fully functional UI and application logic code for data-rich web applications. However, based on RSL alone, one cannot generate data processing (domain logic) code beyond simple CRUD, and data persistence operations [39]. Thus, similarly to other low-code platforms, more complex data processing has to be coded manually.

In this paper, we raise the question of representing more complex domain logic at the level of abstraction used by low-code and requirements-based MDD approaches. Inspiration for responding to this question can be drawn from research on ontologies and knowledge representations. These approaches enable the representation of domain knowledge, independently of any technology and any particular problem domain. Our research aims at investigating how the features of ontology-based domain logic representations, especially their reasoning capabilities, can be applied in the context of LCSD.

We present an extension to the RSL mentioned above, which we call RSL-DL (RSL Domain Logic, see the initial version of the language in our previous work [28]). The new language draws several of its constructs from ontology-based knowledge representation approaches. It then extends and combines them with MDD technologies to provide full code generation capabilities. What is important, RSL-DL allows for the generation of fully operational code directly from general domain rules (descriptions of reality) as required by specific requirements models (use cases and their logic). Hence, such generated back-end code is fully compatible with the front-end code generated from RSL specifications. Through this, we demonstrate a visual extension to a low-code language (RSL) that has the potential of eliminating the need to code (in a traditional sense) complex data processing services.

II. Motivation and related work

The term "low-code" was probably first used in a Forrester report [25] only in 2014. Current studies report numerous industry-grade low-code platforms used extensively by professionals [29]. Sparse research results on LCSD can be supported by previous and current research on Model-Driven Web Engineering [17] which can be seen as a predecessor and now a synonym for LCSD. A study by Wakil and Jawani [38]

[1] https://github.com/smialekm/redseeds

shows that research on MDWE is already quite broad and mature. LCSD and MDWE approaches are typically based on some form of visual notation (language). Such notations offer high-level representations of the flows of interaction between application users and the system under development. A prominent example in the MDWE domain is the Interaction Flow Modeling Language (IFML) [4]. Other examples – in the LCSD domain – are the Business Process Technology (BPT) language [13] and the Mendix notation [12].

LCSD/MDWE systems have limited capabilities regarding the generation of the domain/business logic code, or more broadly – the system's back-end. Current LCSD/MDWE languages can support generation of code for elementary CRUD (Create-Read-Update-Delete) operations [3], [26]. Generation of code for more complex data processing (general Domain Logic - DL) is limited by the information scope of the visual language constructs. In this research, we propose new constructs that significantly extend capabilities to generate complex domain logic code. What is important, these new constructs are domain-agnostic, as contrasted with various domain-specific and often very formalised notations (see, e.g. work by Hinchey et al. [14] and Brito et al. [5]).

Our research is in line with the work by Atkinson et al. [2] that shows significant similarities between ontologies and models. The authors argue that the concept of ontology constitutes a subset of the concept of model. Also, Henderson-Sellers [11] points out that a combination of models and meta-models with domain ontologies is helpful in representing vocabularies for specific problem domains. He argues that modelling languages such as UML can describe domain knowledge, but they need particular extensions to provide adequate reasoning support. Our approach goes in this specific direction, as it extends RSL, which is also an extension to UML. In summary, the above discussions give good motivation for our work, where a modelling language that combines ontology constructs is applied to generate code directly from requirements.

Marrying ontologies with models allows applying model-driven techniques and especially model transformations. Appropriate works include more general discussions on introducing comprehensive formalised propositions on meta-models for ontology languages [23], [9]. An example of such a language is CoCoViLa by Haav, and Ojamma [10]. Our current work can be compared or even contrasted with such approaches, as it introduces a common meta-model for a semantically rich language that can express any problem domain. In this context, an essential feature of our approach is its extensive reliance on inference mechanisms, especially for generating data processing code. It is somewhat similar to business rule engines [8]. However, instead of interpreting them during runtime, it generates code fully integrated with the rest of the system. Similarly, our solution can be compared with the approaches that enable code generation directly from ontologies. Stevenson and Dibson [34] propose a tooling framework for generating Java code from OWL specifications. Another example is work by Völkel and Sure [36] in which Java-based APIs are generated directly from ontologies expressed in RDF Schema.

III. RSL: LOW-CODE AT THE REQUIREMENTS LEVEL

The necessary background to our research on RSL-DL is the Requirements Specification Language and its tooling environment (ReDSeeDS). As we strive to achieve compatibility between these two languages, some aspects of RSL-DL were strongly influenced by RSL. The most important for this purpose are use case scenarios, like the example one shown in Fig. 1. Scenarios consist of sequences of simple subject-verb-object sentences of various types. The most important of them from the point of view of this paper is the 'Query' type sentences. These sentences define actions of the system that are performed on certain data elements.

According to RSL semantics rules [31], we can transform scenarios into code. Fig. 2 presents fragments of an application logic class generated from the presented scenario. The class contains methods for handling user interactions, as specified by the 'Select' sentences in the scenarios. For instance, the sentence no. 1 is translated into the "summarizeSemesterTriggered" method. Contents of these methods reflect consecutive sentences in the scenarios. 'Query' and CRUD type sentences are translated into calls to back-end service operation. For

Name: Summarize semester — Action Type

precondition:

#	Sentence	Action Type
1.	Dean's office employee *selects* summarize semester	Select
2.	System *fetches* students list	Read
3.	System *prepares* semester summarization data	Query
4.	System *shows* semester summarization window	Show
5.	Dean's office employee *selects* accept summarization	Select
6.	System *closes* semester summarization window	Close
7.	System *realizes* semester summarization data	Update
8.	System *shows* semester summarized message	Show

final: success

Fig. 1. Example scenario

```
public class SummarizeSemesterPresenter extends
                AbstractUseCasePresenter {
    // ...
    public void summarizeSemesterTriggered(){
        studentListDTO =
            service.readStudentList();
        semesterSummarizationDataDataDTO =
            service.preparesSemesterSummarizationData
                (pstudentListDTO);
        view.showShowSemesterSummarizationWindow(this);
        pageOpened();
    }

    public void saveFinalGradeTriggered(){
        view.closeSemesterSummarizationWindow();
        pageClosed();
        service.realizesSemesterSummarizationData
            (semesterSummarizationDataDTO);
        view.showSemesterSummarizedMessage();
    }
    // ...
}
```

Fig. 2. Presenter code generated for the use case scenario

```
public class ServiceImpl implements IService {
    // ...
    List<SemesterSummarizationDataItemDTO>
        preparesSemesterSummarizationData
            (List<StudentListItemDTO> studentListDTO)
    {}

    void realizesSemesterSummarizationData
        (List<SemesterSummarizationDataItemDTO>
            semesterSummarizationDataDTO)
    {}
    // ...
}
```

Fig. 3. Backend access code generated for the example scenario

instance, the sentence no. 7 is transformed into a call to the “realizesSemesterSummarizationData” service. UI presentation sentences are translated into calls to the View layer. A more detailed discussion of the rules and generated code, including code of the View layer, is presented elsewhere [32].

The relevant parts of the back-end service code generated from the RSL scenario is presented in Fig. 3. It contains an interface implementation with empty methods. The operation parameters are determined from scenario sentences before the appropriate calls. In further sections, we will present the syntax and semantics of RSL-DL that will fill the currently empty method bodies.

IV. RSL-DL SYNTAX

The syntax of RSL-DL aims to represent all information important from the point of view of code generation. It includes detailed dependencies between individual domain elements and proper definitions of the elements themselves. Fig. 4 presents an elementary example of concrete syntactic elements of the language. It contains definitions of four “Identity” type entity notions (student, course, partial grade, weighted grade), describing concrete objects in the specific problem domain. Notions can also have conditions, and in our example, we can see one kind of condition: “inheritance”. Thus, the condition for the “partial grade” notion is that it must follow all the rules for the “weighted grade” notion. In addition to entity notions, we can define property notions, like “grade weight” in Fig. 4. This kind of notions define concrete atomic values and can be used as attributes of other notions, which can be indicated by ‘attribute links’ (lines with a diamond shape).

Relationships in RSL-DL (see “grading” in Fig. 4) are represented by hexagons and can link many notions. To some extent, this syntax resembles that of UML’s n-ary associations. In our example, the relationship is of type “Data-Based Reference” which is a basic type that reflects the situation where references between objects are contained in their data (e.g. in their attributes). In our current example, “student” and “course” contribute to the relationship “grading” that results with a “partial grade”. Arrow directions distinguish between types of notion participations in the specific relationship. Since a given student can have many partial grades in a course, then the particular participation is marked as ‘multiple’.

The abstract syntax for the above-presented core language elements is presented in Fig. 5a using the MOF notation [22]. The meta-class “DLNotion” represents notions and the meta-class “DLRelationship” represents dependencies between them. Concrete participations of notions in relationships are represented by the meta-class “DLRelationshipParticipation”. The meta-model contains two types of such participations – standard and auxiliary. Standard ones correspond to the main subjects of relationships and are denoted with solid arrows in concrete syntax. Auxiliary ones point to elements that define relationship contexts and are denoted with dashed lines. For example, one could use a relationship context to indicate which object should be used when computing values based on that object’s attributes. In this case, the attributes participate through standard participations, and their ‘parent’ participates through auxiliary participation.

Besides notions, there is a special kind of relationship participants – primitives (“DLPrimitive” meta-class). These elements define general concepts that do not have concrete instances. Examples of such primitives in RSL-DL are “current date”, “number Pi” and “Planck constant”.

As indicated above, notions can have types. The first one (“identity”) was explained in the example above. The “template” type indicates templates that can be used to simplify defining other notions. These two types correspond approximately to concrete and abstract classes of e.g. UML. Two other types define notions whose representatives’ (objects’) roles can change during their lifetime. It is inspired by ontology-based inference engines with their capabilities to “discover” object types or change them dynamically. The “inferred role” type indicates roles that can be inferred, e.g. from various status attributes of an object. The “assigned role” type indicates roles that can be explicitly changed during the lifetime of an object.

More details related to the syntax for notions are shown in Fig. 5b. The “DLProperty” meta-class is used to denote notions with concrete atomic values or value sets. The “DLEntity” meta-class denotes more complex notions that cannot be reduced to single values. The “DLAttributeLink” meta-class allows indicating attribute dependencies between notions. Such links can be marked as ‘derived’, which means that their values need to be inferred from other notions. An important type of notion features are conditions (“DLCondition” meta-class). Their role is to further detail notion characteristics. Apart from the previously described ‘inheritance condition’, two additional condition types exist. The ‘identity condition’ type defines conditions that have to be fulfilled for a given notion’s

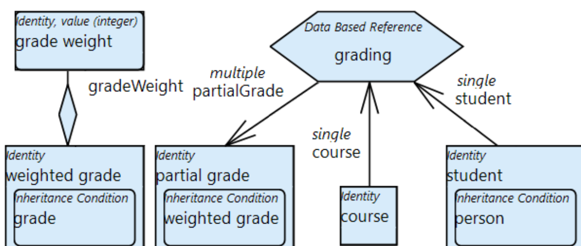

Fig. 4. RSL-DL concrete syntax example

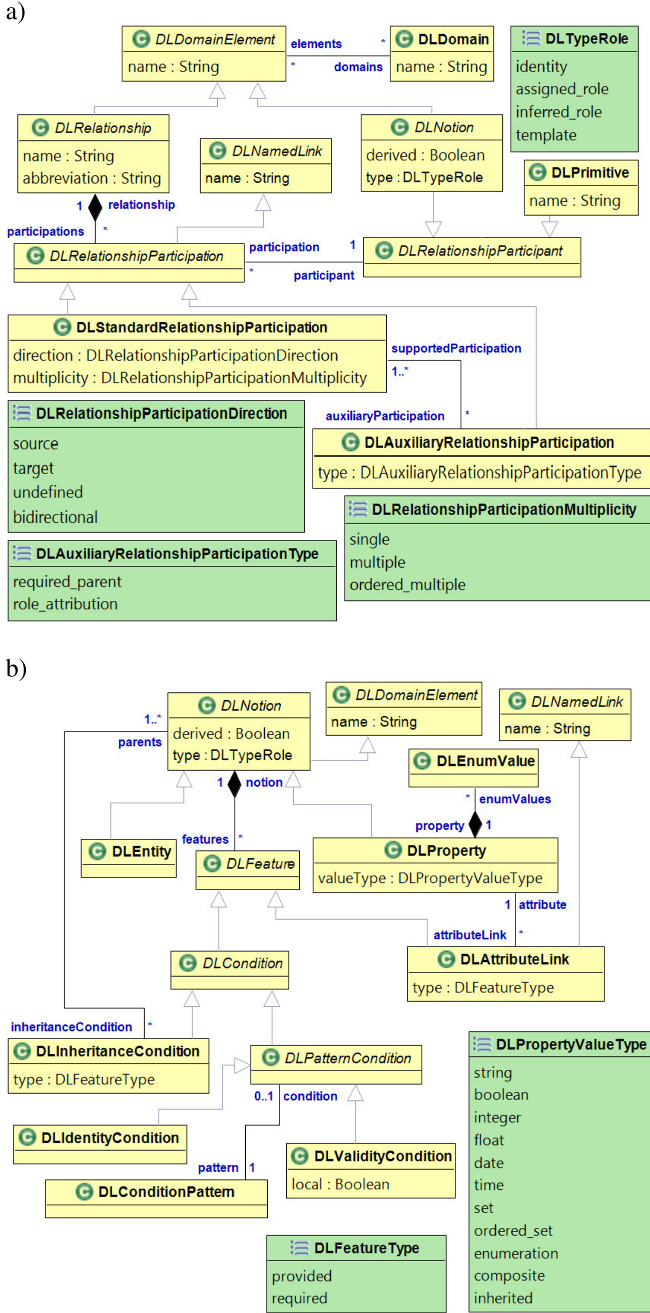

Fig. 5. Meta-model fragments for core RSL-DL elements (a), and for notions (b)

object to make sense. The ‘validity conditions’ type defines conditions that denote the correctness of a given notion’s object. In general, there can exist objects that meet appropriate identity conditions but do not meet validity conditions and thus are treated as invalid but belonging to the given notion. We should note that conditions do not include graphical links to other model elements. It is due to their potential complexity and interweaving. Thus, for instance, inheritance was not represented using a simple arrow as in UML.

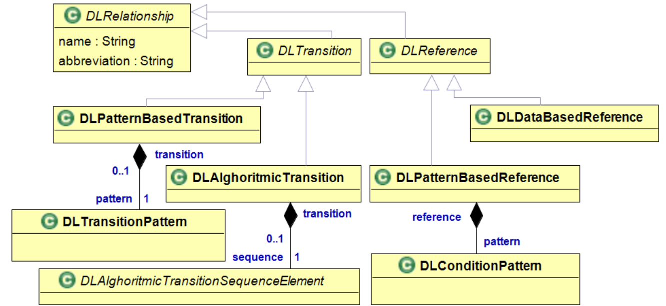

Fig. 6. Meta-model details for relationships

Fig. 6 presents the hierarchy of relationships found in RSL-DL. From the conceptual point of view, two main types of dependencies exist between notions in the problem domain that are significant for code generation. It is reflected in RSL-DL through dividing relationships into two categories: transitions (“DLTransition”) and references (“DLReference”). Transitions describe how to obtain notion objects based on other notion objects. References describe specific roles played by objects in relation to other objects. Both relationship categories are further divided based on how they are defined. ‘Transitions’ can be described using simple rules (“DLPatternBasedTransition”) or algorithms consisting of many steps (“DLAlgorithmicTransition”). ‘References’ can be described using rules that define certain conditions (“DLPatternBasedReference”) or take the form of the previously described data-based references (“DLDataBasedReference”). This division of references is inspired by the division into fact and rule spaces found in ontologies. In practice, of all these relationships, the algorithmic transitions are not preferred as they are not fully declarative and thus arguably less usable [15], [37].

Figs. 5b and 6 contain two additional elements: “DLTransitionPattern” and “DLConditionPattern”. Each instance of these two meta-classes contains a string with a textual condition, a specific condition type and optionally – the condition’s subject link. The syntax of the condition is expressed in a language based on the notation used in the Symja library [18]. Some examples of several types of these patterns are given in section VI in descriptions of the generated code.

V. Translational Semantics Rules

Of the many approaches to define semantics for RSL-DL (treated as a programming language [30]) we choose the translational method, which is more in line with the model-driven paradigm. This approach defines rules that translate specific patterns of RSL-DL constructs into fragments of Java code. Each rule has an informal textual description and is formalised as a procedure in the MOLA graphical model transformation language [16].

Full specification of semantics for RSL-DL consists of 16 translational rules (all the details can be accessed in the Supplement[2]). The first ten rules define the generation

[2]https://github.com/smialekm/redseeds/tree/main/RSL-DL

of the target Java class structure, including their fields and method signatures. These rules depend only on the structure of notions and relationships between them, found in a particular RSL-DL model. The following two rules additionally use an inference engine and are used to generate method bodies. Rules 13-16 further add to the generation of method bodies. They use a symbolic computation library to transform Symja-based formulas found in pattern condition expressions into the contents of method bodies. The results are used directly or as part of a loop or a condition depending on the pattern type. In summary, each 'non-trivial' notion in the source RSL-DL model produces two Java classes. One class represents (in simplified terms) a data transfer object (DTO) corresponding to the given notion. The other class is a utility class that holds various data handling methods. These classes are appropriately amended with CRUD and condition-related operations. The "DTO" classes are also organised in an appropriate inheritance hierarchy. Additionally, supportive classes are created for all the relationships in the model. These classes contain methods that return objects participating in relevant relationships.

As introduced above, all the rules are formalised using MOLA procedures. MOLA uses a declarative-imperative visual syntax presented in Fig. 8 and 9. Its imperative flow definition is based on a notation resembling activity diagrams in UML. Arrows denote control flow. Iteration 'actions' are denoted with thick black frames. Rule 'actions' constitute the declarative part of the language. Each rule contains a query on objects expressed through a diagram resembling a UML object diagram combined with a MOF meta-model diagram. Black solid lines denote queried objects, while red dashed lines denote created objects. More details and a tutorial can be found in the MOLA handbook [1].

For brevity, we will limit our presentation of rule formalisation to only rule 11. To implement it, we need to use a dedicated inference engine, implemented as part of this work. The engine processes queries derived from 'Query' type scenario sentences (see again Fig. 1). For each such query, it produces a sequence of inference rules, where each of the rules is based on domain elements defined within an RSL-DL model. The appropriate sequences can be represented with a meta-model shown in a simplified form in Fig. 7. The meta-model uses a structure of nested "Rule" meta-classes to reflect appropriate sequences of inference invocations needed to solve specific problems. Each "Rule" points to a domain

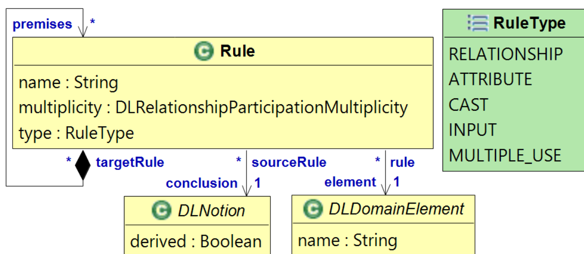

Fig. 7. Inference rule meta-model

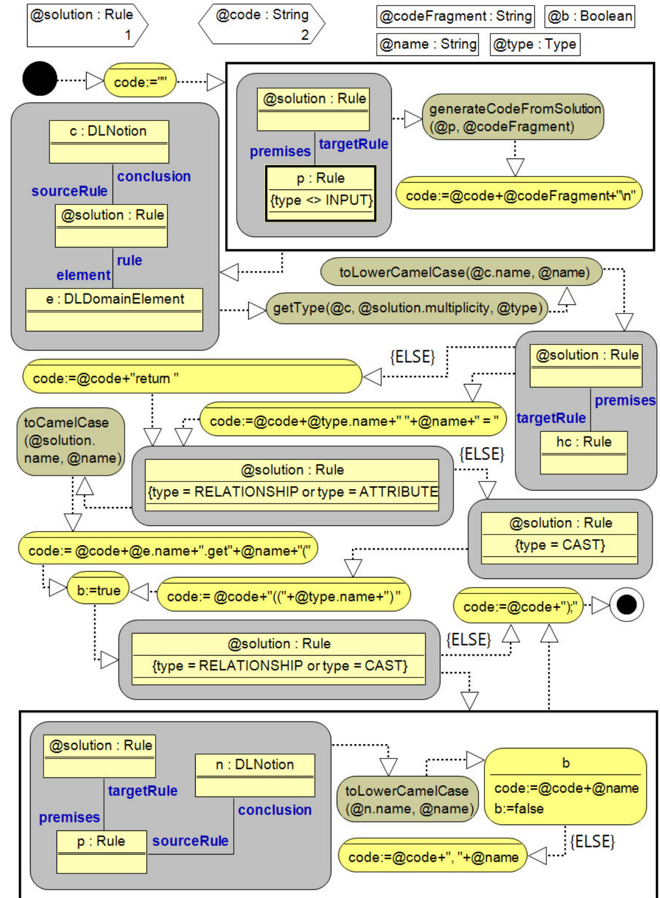

Fig. 8. Algorithm for generating the method body in Rule 11

element ("element") that is the basis for generating a specific method according to one of the previous rules. The "type : RuleType" meta-attribute defines the concrete type of such generation. This "Rule" meta-class also points to a 'conclusion' that constitutes a specific notion reflecting objects being the inference results. Furthermore, the given rule's 'premises' constitute other rules, preceding this rule in the rule sequence. Base premises that reflect query parameters are represented as additional 'artificial' rules.

The algorithm that generates the respective sequence of method invocations from the inference rule structure is presented in Fig. 8. It starts from invoking itself ('generateCodeFromSolution') recursively for all the 'premises' of the current rule and joining code generated from these premises. If the current element is used as a 'premise' in other rules, a proper variable is declared based on the object corresponding to the rule's 'conclusion'. Otherwise, a 'return' statement is generated. In both cases, the way to obtain the assigned or the returned value depends on the type of the rule. In most cases, such a value is obtained by invoking an appropriate 'get' method. This method retrieves an object that corresponds to the 'conclusion' and is contained in the class derived from the rule's 'element'. Besides, the 'get' method's call accepts

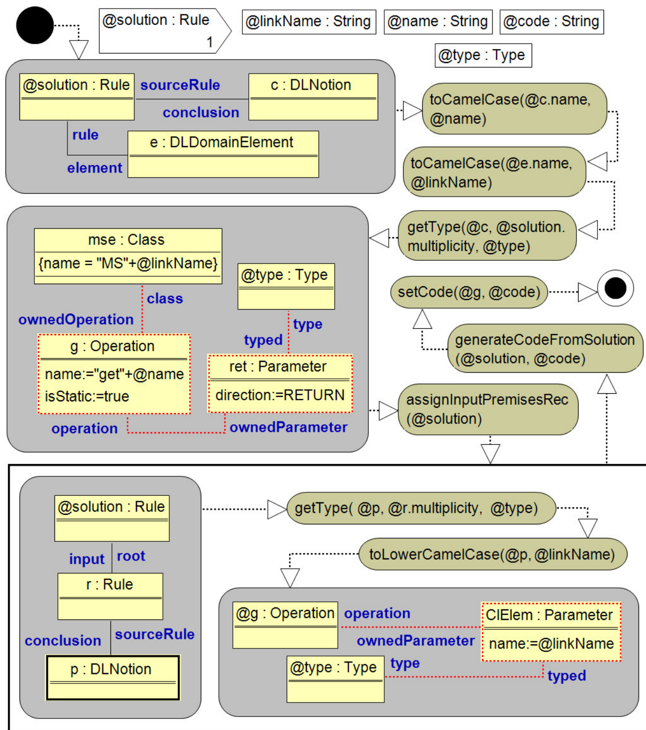

Fig. 9. Formalization of transformation rule 11

parameters that correspond to the rule's premises.

The actual formalisation of Rule 11 that uses the above algorithm is presented in Fig. 9. It defines a procedure for creating a method, where the method's contents are generated with the algorithm. The procedure starts by creating a method with the name based on the 'conclusion' of the final rule and having the prefix "get". This method is placed in the class derived from the domain element pointed to by this final rule. The return type of this method again corresponds to the rule's 'conclusion', and the method's parameters are based on the 'premises' within the whole rule sequence.

VI. Case Study

This section will present a selected fragment of a more extensive case study that illustrates several important uses of RSL-DL. The case study refers to the functional requirements specification presented in Section III. Here we will concentrate only on presenting examples of the various concrete RSL-DL language constructs, generated domain logic code, and references to application logic code from Section III.

Fig. 10 involves constructs for checking specific conditions. The model contains elements that describe information related to checking whether the given student is eligible to get a registration for the next semester. It consists of three basic notions – 'student', 'course' and 'final grade'. All of them are connected by the "final grading" relationship, which is a data-based reference. It indicates that information about concrete dependencies between representatives of these notions is stored in some data objects. Finally, we define the 'student to

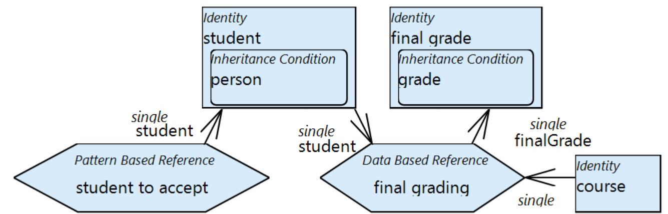

Fig. 10. RSL-DL model defining eligibility of students to be registered for the next semester

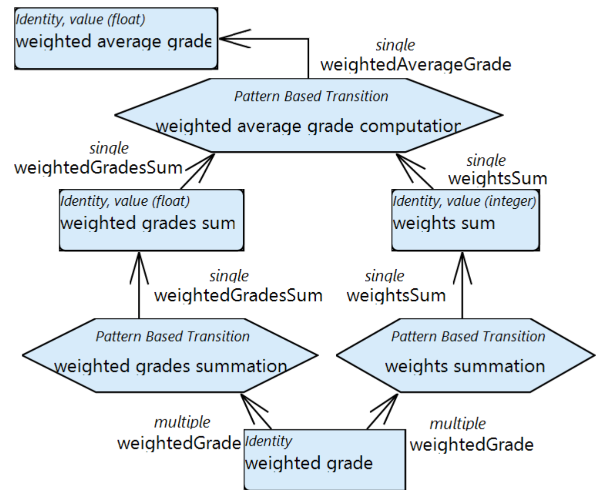

Fig. 11. RSL-DL model containing knowledge about computing of weighted average grades

accept' relationship that is used to define conditions about students' eligibility to be registered for the next semester. The concrete condition embedded in this relationship (not shown here) requires that all the final grades for the student's courses have a value of at least 3 (minimum passing level). The relationship has only one participant – the student, that participates as its target.

Fig. 11 involves constructs for computing values. It contains the 'weighted average grade computation' relationship that contains a transition pattern with an equation that computes the weighted average grade. This equation requires two other values represented by the notions 'weighted grades sum' and 'grades sum'. Therefore, the model contains also transitions that allow for the computation of these two values.

The final part of the presented model fragment involves constructs for modifying complex objects and is shown in Fig. 12. The whole modification is handled by the 'summarize student after semester' transition, which requires two other transitions: "accept student", and "fail student". These transitions will be used interchangeably, depending on the fulfilment of a condition. This condition refers to the 'student to accept' relationship presented in Fig. 10. If the student meets the 'student to accept' relationship, the 'accept student' transition is invoked, while in the opposite case, the 'fail

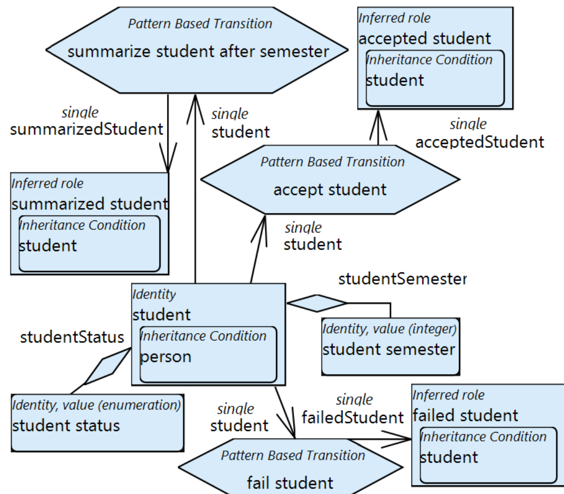

Fig. 12. RSL-DL model containing knowledge about the rules for promoting students to the next level of studies

```
public class MSStudentToAccept {
    public static boolean checkStudentToAccept(
        IMStudent student){
            for (IMFinalGrade $_iter:
                student.getFinalGrades())
                    if (!$_iter.getGradeValue()>=3)
                        return false;
            return true;
        }
    public static List<IMStudent> getStudents(){
            List<IMStudent> students
                = MSStudent.getStudents();
            List<IMStudent> result
                = new ArrayList<IMStudent>();
            for (IMCourse $_iter:students)
                if (checkStudentToAccept($_iter))
                    result.add($_iter);
            return result;
        }
}
```

Fig. 13. Code for checking students' registration eligibility

student' transition is invoked.

The next step in our case study is to generate code. The above specification in RSL-DL was formulated within a dedicated RSL-DL tool. The tool also includes a code generation engine that implements all the rules introduced in the previous section. Here we will present some key fragments of code generated from the above-presented excerpts of the dean office model.

Fig. 13 shows code generated on the basis of the model from Fig. 10. The actual class (MSStudentToAccept) is generated per rule no. 9 (generate classes and static methods from relationships). It is a supportive class that corresponds to the 'student to accept' relationship and contains only static methods. The 'checkStudentToAccept' method was generated based on rule no. 10 (generate existence checking methods). It checks for the eligibility of a given Student to be accepted for the next semester. The student is passed as the parameter of this method. It can be noted that the type of this parameter ('IMStudent') is the class corresponding to the 'student'

```
public class MSWeightedAverageGradeComputation {
    public static double
        getWeightedAverageGrade(double weightedGradesSum,
                int weightsSum){
            return weightedGradesSum/weightsSum;
        }
    public static double
        getWeightedAverageGrade(List<IMPartialGrade> partialGrades){
            List<IMWeightedGrade> weightedGrades
                = new List<IMWeightedGrade>(partialGrades);
            double weightedGradesSum
                = MSWeightedGradesSummation
                    .getWeightedGradesSum(weightedGrades);
            int weightsSum = MSWeightsSummation
                    .getWeightsSum(weightedGrades);
            return getWeightedAverageGrade(weightedGradesSum,
                    weightsSum);
        }
}
```

Fig. 14. Code for computing weighted average grades

notion, generated according to rule no. 1 (generate classes from notions).

The method returns a logical value reflecting the result of the eligibility check. The actual check is based on the contents of the condition pattern in the 'student to accept' relationship (not shown in Fig. 10). This pattern is defined through three values: 1) formula 'gradeValue($)>=3', 2) type 'universal quantification', and 3) subject link 'finalGrading(student)' that relates to the 'final grading' relationship. Note that the '$' sign denotes the target of the 'final grading' relationship, which is the 'final grade' notion. The above formula was transformed into the appropriate 'if' statement in Fig. 13. The 'for' loop is generated based on rule no. 13 (generate condition checking code from condition patterns), considering the above pattern type. This way, the eligibility check is done for all the grades of a particular student.

The second method of this class ('getStudents') applies the above eligibility check to all the students. This method was generated based on rule no. 14 (generate object filtering code from condition patterns), and it filters out all the subjects (here: students) that fulfil an appropriate condition pattern (here: student eligibility check). We should also note that the code for obtaining the list of all students was generated using rule no. 16 (generate auxiliary code).

Fig. 14 shows code generated on the basis of the model from Fig. 11, using data structured according to Fig. 4. This time, the situation in somewhat different to that in the previous code fragments. This part of the code results from answering a query that asks to compute the 'weighted average grade' for the given set of grades. It contains two overloaded methods ('getWeightedAverageGrade'). The second one (with the 'List' parameter) is generated according to rule no. 11. It accepts a list of partial grades and produces a specific average value. The method contains a sequence of method calls that reflect the sequence of inference rules returned by the inference engine (see Fig. 8). The first one of the overloaded methods is called from the second one. Its signature was generated according to rule no. 9. Its body was based on translating a transition pattern with the 'simple' formula 'weightedGradesSum/weightsSum'

```
public class MSSummarizeStudentAfterSemester {
    public static List<IMSummarizedStudent> getSummarizedStudents
        (List<IMStudent> students) {
        List<IMSummarizedStudent> result =
                new ArrayList<IMSummarizedStudent>();
        for (IMStudent $_iter:students)
            result.add(getSummarizedStudent($_iter));
        return result;
    }
    public static IMSummarizedStudent getSummarizedStudent
        (IMStudent student) {
        if (MSStudentToAccept.checkStudentToAccept(student))
            return new MSummarizedsStudent(MSAcceptStudent.
                    getAcceptedStudent(student));
        return new MSummarizedsStudent(MSFailStudent.
                getFailedStudent(student));
    }
}
```

Fig. 15. Code for determining student's eligibility for promotion to the next semester

```
public class ServiceImpl implements IService {
    // ...
    List<SemesterSummarizationDataItemDTO>
        preparesSemesterSummarizationData
            (List<StudentListItemDTO> studentListDTO)
    {
        return MSummarizeStudentAfterSemester.
            getSummarizedStudents(studentListDTO);
    }
    // ...
}
```

Fig. 16. Additional back-end access code

according to rule no. 15 (generate code from transition patterns).

Fig. 15 shows code generated on the basis of the model from Fig. 12. This time we can see only one method ("getsummarizedStudent") generated according to rule no. 9. The method's body is generated on the basis of a 'mapping' transition pattern with the formula 'studentToAccept(student); acceptStudent(student); failStudent(student)'. Formulas for this type of transition patterns are composed of three sections: a 'simple' condition pattern formula and two 'simple' transition pattern formulas (the first one used when the condition is true, and the second one otherwise). Here, the condition pattern refers to the 'student to accept' relationship shown in Fig. 10. Thus, the current method calls the 'checkStudentToAccept' method shown in Fig. 13. Depending on its result, it calls one of two methods resulting from transforming the 'accept student' and 'fail student' relationships. The other method in this class invokes the above described one over a set of appropriate objects (list of students).

Finally, relevant code fragments as presented above, can be now applied to fill-in appropriate empty methods of the back-end service class (see Fig. 3). In our example this pertains to the method that corresponds to a non-CRUD operation. Appropriate additional code is presented in Fig. 16. It contains a simple call to the operation presented in Fig. 15. It is worth noting that such operations are optimised to use only required parameters.

VII. Language validation and discussion

To initially validate the presented language, we have used two different approaches. Their aim was to assess certain aspects of the language's usability: understandability and operability. The first approach was to determine language comprehension by its first-time users. It consisted in testing language proficiency, following a brief introduction to the language. The second approach was to determine efficiency of language usage by the users with various experience levels. In both cases, we have used a specially developed RSL-DL editor, used in conjunction with the ReDSeeDS environment.

A. Validation of understandability

The first validation study was conducted with a group of post-graduate computer science students attending the "Model-Driven Software Development" course at the Warsaw University of Technology. The course curriculum included classes on the design and usage of various Domain-Specific Languages. The study was thus well aligned with the aim to acquaint the students with this topic.

The setup of the study was as follows. First, the students attended two lab sessions (four class hours) where they were presented with the RSL and the ReDSeeDS tool. Note that prior to this, the students had no experience with Software Language Engineering but have attended a parallel lecture where they were introduced with the fundamentals of meta-modelling. Next, the students were presented with a brief, one-hour introduction to the language. Then, they have spent two hours solving simple exercises using the aforementioned RSL-DL editor. After this, the students were presented with correct solutions to the exercises. Finally, the students were asked to answer 12 questions in an online questionnaire. All of the questions were single-choice, and referred to specific RSL-DL diagrams. Each question had four possible answers. The first eight questions were related to the understanding of the language syntax, the next three related to language usage, and the final one checked more nuanced usage of the language related to its declarative nature. The students were given one class hour (45 minutes) to finish the questionnaire, but most of them have finished in less than 20 minutes.

The results of the study consist in 42 replies to the questionnaire. The average of correct answers in the whole questionnaire was 69%. For syntax understanding (the first eight questions), it was 75%, for usage understanding (the next three questions), it was 60%, and for the last question it was 40%. Detailed results, together with the question contents are provided in the Supplement.

The relatively low result in the case of the last question can be explained by its advanced nature, going beyond the explanations given to the students. Thus, the 40% can be seen as an unexpectedly good result. It is also worth noting that relatively low percentages of correct answers were associated with questions about differentiation between inferred roles and assigned roles (questions no. 3, 5 and 9). These results were similar to the case of the last question. Further research is

TABLE I
RESULTS OF THE OPERABILITY STUDY

Participant	Task	RSL-DL time	Java time
Author	No. 1	3:00 min.	5:00 min.
Ph.D. Student	No. 1	9:00 min	12:00 min.
Undergrad. Student	No. 1	13:35 min	4:50 min.
Author	No. 2	1:45 min.	3:40 min.
Ph.D. Student	No. 2	4:00 min	10:00 min.
Undergrad. Student	No. 2	16:45 min	5:30 min.
Author	No. 3	4:30 min.	6:00 min.
Ph.D. Student	No. 3	15:00 min	15:00 min.
Undergrad. Student	No. 3	12:25 min	6:20 min.

needed to determine if that was caused by insufficient explanations (this aspect was under-represented in the exercises) or inherent difficulties caused by the language design. In summary, the overall results of this study indicate that the language is comprehensible even after a very short introduction. However, a more thorough validation with statistical analysis is needed to confirm this, and can be seen as future work.

B. Validation of operability

The second validation study was conducted with a group of three software developers with different programming skills. The first person is one of the language authors and thus has very good knowledge of RSL-DL. At the same time, he is an experienced Java programmer. The second person is a Ph.D. student with wide general computer science knowledge and average Java programming experience. The third person is an undergraduate student with more narrow CS knowledge but with relatively high experience in Java programming. The students were not involved in the development of RSL-DL and had no previous knowledge of it.

The study consisted in comparison of coding efficiency and was based on solving specific problems. The setup of the study was as follows. First, the study participants were presented with a 1.5 hour long introduction of RSL-DL and its editor. This included the presentation of three problems: calculation of square mean error (no. 1), calculation of definite integrals (no. 2), and calculation of VAT for product lists (no. 3). The problem formulations involved appropriate formulas and are presented in detail in the Supplement. Next, the participants were supplied with artefacts generated from appropriate RSL specifications (use cases with scenarios) by the ReDSeeDS system. These consisted of pre-initialised RSL-DL models (just the notions) and code skeletons (Data Transfer Objects and method signatures) in Java.

The goal of the participants was to fill-in the provided artefacts to complete domain logic functionality. To prevent from negative bias, the participants were asked to solve the problems using RSL-DL first, and only then to solve them in Java. The participants were also asked to measure time spent on all the tasks. The results of these measurements are given in Table I.

Comparison of times for the three study participants can be treated as rather anecdotal evidence but they give some insight on the productivity of developing domain logic (backend) code with RSL-DL. As it can be noticed, productivity of RSL-DL development vs. Java development is significantly higher for an experienced RSL-DL user. Also, a less experienced Java programmer (the Ph.D. student) had certain productivity gains. On the other hand, a very experienced Java programmer (the undergraduate student) had performed much better using a traditional programming language. Thus, it can be argued that as general knowledge of developers and their RSL-DL skills raise - productivity gains tend to be significant. It can also be argued that RSL-DL has the potential for extending productivity gains for less experienced programmers. Still, this argumentation has to be acknowledged through a more thorough experimentation with a larger participant scope. This can be seen as future work.

VIII. SUMMARY AND FUTURE WORK

In this paper, we have shown that declarative knowledge representations can be used to produce imperative (3GL) back-end code with non-trivial domain logic. Moreover, this code can be interfaced with front-end code produced from low-code specifications that use formalised requirements models (use cases, scenarios). To achieve this, we have used a combination of techniques from model-driven development and ontology-based inference. In this environment, most "programming" activities could be made using a high-level visual language. Thus, programming becomes equivalent to specifying models that define various aspects of the system and its problem domain. An RSL-DL conceptual model can be treated – in fact – as a high-level program that can be executed immediately after compiling it into eg. Java and then – executable code. Moreover, RSL-DL models can be seen as "ontologies as code" [19] and a step towards a "fifth generation language" as postulated by Thalheim and Jaakkola [35].

We see two main areas where our approach can benefit software development: reduction of complexity and increased reuse. The first area is in line with the general goals of the low-code movement – to offer means for reducing accidental (technological) complexity in favour of concentrating on the essential (e.g. domain) complexity (see early insights on this by Brooks [6]). A thorough comparison of complexity between RSL-DL and traditional programming languages, and analysis of reusability can be seen as interesting areas of future work.

Another area for future work is the analysis of RLS-DL usability as a low-code language. Generally, it can be expected that better usability is assured through declarative characteristics of the language (see appropriate comparative analyses [15], [37]). This is in line with our initial studies presented in the previous section. However, to fully support this claim, more extensive experimentation should be conducted.

Other areas which we plan to investigate in the future include better integration with requirements processing mechanisms. One aspect of this is the application of natural language processing. Here, valuable insights can be drawn from the concept of naturalistic programming [24]. This concept postulates the use of natural language elements to design programming languages that are more expressive from the programmers'

point of view. Another interesting approach in this area is that of Mefteh et al. [20]. In this approach, natural language scenarios are transformed into constrained language models expressed in RSL. On the other hand, we also plan to integrate our approach with existing approaches to generate CRUD operations and database schemas directly from requirements models expressed in RSL [39].

As a final remark we address the question of creating a distinct new language as opposed to using an extension to the existing language (e.g. a UML profile). In our opinion, from the practical point of view, both approaches can be seen as equivalent. However, creating a language from scratch favours solutions that go beyond the "beaten path".

REFERENCES

[1] *The MOLA Language Reference Manual Version 2.0 final*, 2007.

[2] Colin Atkinson, Matthias Gutheil, and Kilian Kiko. On the relationship of ontologies and models. In *Proc. 2nd Workshop on Meta-Modelling, WoMM 2006*, pages 47–60, 2006.

[3] Fábio Paulo Basso, Raquel Mainardi Pillat, Toacy Cavalcante Oliveira, Fabricia Roos-Frantz, and Rafael Z. Frantz. Automated design of multi-layered web information systems. *Journal of Systems and Software*, 117:612–637, 2016.

[4] Marco Brambilla and Piero Fraternali. *Interaction flow modeling language: Model-driven UI engineering of web and mobile apps with IFML*. Morgan Kaufmann, 2014.

[5] Isabel Sofia Brito, Joao Paulo Barros, and Luis Gomes. From requirements to code (Re2Code) – a model-based approach for controller implementation. In *2016 IEEE 14th International Conference on Industrial Informatics (INDIN)*, pages 1224–1230, 2016.

[6] Frederick P Brooks. No silver bullet: Essence and accidents of software engineering. *IEEE Computer*, 20(4):10–19, April 1987.

[7] Jordi Cabot. Positioning of the low-code movement within the field of model-driven engineering. In *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*, MODELS '20, 2020.

[8] Malcolm Chisholm. *How to build a business rules engine: extending application functionality through metadata engineering*. Morgan Kaufmann, 2004.

[9] Dragan Gašević, Dragan Djurić, and Vladan Devedžić. *Model Driven Engineering and Ontology Development*. Springer, 2009.

[10] Hele Mai Haav and Andres Ojamaa. Semi-automated integration of domain ontologies to DSL meta-models. *International Journal of Intelligent Information and Database Systems*, 10(1/2):94–116, 2017.

[11] Brian Henderson-Sellers. Bridging metamodels and ontologies in software engineering. *Journal of Systems and Software*, 84(2):301–313, 2011.

[12] Martin Henkel and Janis Stirna. Pondering on the key functionality of model driven development tools: The case of Mendix. In *International Conference on Business Informatics Research*, pages 146–160. Springer, 2010.

[13] Henrique Henriques, Hugo Lourenço, Vasco Amaral, and Miguel Goulão. Improving the developer experience with a low-code process modelling language. In *21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*, pages 200–210, 2018.

[14] Michael G Hinchey, James L Rash, and Christopher A Rouff. Requirements to design to code: Towards a fully formal approach to automatic code generation. Technical report, NASA, 2005.

[15] Ahmad Jbara, Arieh Bibliowicz, Niva Wengrowicz, Natali Levi, and Dov Dori. Toward integrating systems engineering with software engineering through object-process programming. *International Journal of Information Technology*, pages 1–35, 2020.

[16] Audris Kalnins, Janis Barzdins, and Edgars Celms. Model transformation language MOLA. *Lecture Notes in Computer Science*, 3599:62–76, 2004.

[17] Nora Koch, Santiago Meliá-Beigbeder, Nathalie Moreno-Vergara, Vicente Pelechano-Ferragud, Fernando Sánchez-Figueroa, and J Vara-Mesa. Model-driven web engineering. *Upgrade-Novática Journal (English and Spanish)*, 2:40–45, 2008.

[18] Axel Kramer. Symja library-Java symbolic math system, 2018. last accessed May 2020.

[19] Beatriz Franco Martins. The OntoOO-method: An ontology-driven conceptual modeling approach for evolving the oo-method. In *Advances in Conceptual Modeling*, pages 247–254, 2019.

[20] Mariem Mefteh, Nadia Bouassida, and Hanene Ben-Abdallah. Towards naturalistic programming: Mapping language-independent requirements to constrained language specifications. *Science of Computer Programming*, 166:89–119, 2018.

[21] Wiktor Nowakowski, Michał Śmiałek, Albert Ambroziewicz, and Tomasz Straszak. Requirements-level language and tools for capturing software system essence. *Computer Science and Information Systems*, 10(4):1499–1524, 2013.

[22] Object Management Group. *Meta Object Facility (MOF) Core Specification, version 2.5.1, formal/2019-10-01*, 2019.

[23] Fernando Silva Parreiras, Steffen Staab, Simon Schenk, and Andreas Winter. Model driven specification of ontology translations. In *ER'08*, volume 5231 of *Lecture Notes in Computer Science*, pages 484–497, 2008.

[24] Oscar Pulido-Prieto and Ulises Juárez-Martínez. A survey of naturalistic programming technologies. *ACM Comput. Surv.*, 50(5), 2017.

[25] Clay Richardson, John R Rymer, Christopher Mines, Alex Cullen, and Dominique Whittaker. New development platforms emerge for customer-facing applications, 2014. Forrester report.

[26] Roberto Rodriguez-Echeverria, Juan C Preciado, Alvaro Rubio-Largo, José M Conejero, and Alvaro E Prieto. A pattern-based development approach for Interaction Flow Modeling Language. *Scientific Programming*, 2019, 2019.

[27] Davide Di Ruscio, Dimitris Kolovos, Juan de Lara, Alfonso Pierantonio, Massimo Tisi, and Manuel Wimmer. Low-code development and model-driven engineering: Two sides of the same coin? *Software and Systems Modeling*, pages 1–10, jan 2022.

[28] Kamil Rybiński and Rafał Parol. RSL-DL: Representing domain knowledge for the purpose of code generation. In *Software Engineering: Challenges and Solutions*, pages 61–73. Springer, 2017.

[29] A. Sahay, A. Indamutsa, D. Di Ruscio, and A. Pierantonio. Supporting the understanding and comparison of low-code development platforms. In *46th Euromicro Conference on Software Engineering and Advanced Applications*, pages 171–178, 2020.

[30] Kenneth Slonneger and Barry L Kurtz. *Formal Syntax and Semantics of Programming Languages*. Addison-Wesley, 1995.

[31] Michał Śmiałek, Norbert Jarzebowski, and Wiktor Nowakowski. Runtime semantics of use case stories. In *Visual Languages and Human-Centric Computing (VL/HCC), 2012 IEEE Symposium on*, pages 159–162. IEEE, 2012.

[32] Michał Śmiałek and Wiktor Nowakowski. *From Requirements to Java in a Snap: Model-Driven Requirements Engineering in Practice*. Springer, 2015.

[33] Michal Smialek and Tomasz Straszak. Facilitating transition from requirements to code with the ReDSeeDS tool. In *Requirements Engineering Conference (RE), 2012 20th IEEE International*, pages 321–322, 2012.

[34] Graeme Stevenson and Simon Dobson. Sapphire: Generating java runtime artefacts from OWL ontologies. In *Progress in Pattern Recognition, Image Analysis, Computer Vision, and Applications*, pages 425–436. Springer, 2011.

[35] Bernhard Thalheim and Hannu Jaakkola. Model-based fifth generation programming. In *Information Modelling and Knowledge Bases XXXI*, pages 381–400. IOS Press, 2020.

[36] Max Völkel and York Sure. RDFReactor-from ontologies to programmatic data access. In *Poster Proceedings of the Fourth International Semantic Web Conference*, 2005.

[37] Petri Vuorimaa, Markku Laine, Evgenia Litvinova, and Denis Shestakov. Leveraging declarative languages in web application development. *World Wide Web*, 19(4):519–543, 2016.

[38] Karzan Wakil and Dayang NA Jawawi. Model driven web engineering: A systematic mapping study. *e-Informatica Software Engineering Journal*, 9(1):107–142, 2015.

[39] Nassima Yamouni Khelifi, Michał Śmiałek, and Rachida Mekki. Generating database access code from domain models. In *2015 Federated Conference on Computer Science and Information Systems*, pages 991–996, 2015.