Computer Science and Information Systems 21(3):685–724 https://doi.org/10.2298/CSIS230102024R

Low-Code Development Using Requirements and Knowledge Representation Models*

Kamil Rybiński[1] and Michał Śmiałek[1]

Warsaw University of Technology
pl. Politechniki 1, Warsaw, Poland
{kamil.rybinski, michal.smialek}@pw.edu.pl

Abstract. Low-Code Development is a new software development paradigm which is typically used to generate web applications from high-level visual notations. These notations allow to express the User Interface and Application Logic (user-system interactions) in a way which is understandable by the end-users. Using certain model-driven approaches, the low-code environments allow for generating the front-end layer and basic CRUD operations in the back-end. Yet still, non-standard domain logic (data processing) operations still necessitate the use of traditional programming. In this article we present a visual language, called RSL-DL that can be used to represent such non-standard domain logic in a visual form. It allows to capture 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. The development environment of RSL-DL enables fully automatic generation of domain logic code by reasoning over and reusing domain knowledge. The environment includes a dedicated model editor and a transformation engine. 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 article presents a list of these rules in an informal way, and then introduces their formalisation using a graphical transformation notation. The RSL-DL environment includes an inference engine that enables processing queries to domain models and selecting appropriate invocations to generated code. It was also initially validated through studies that involve understandability, operability and complexity assessment. Based on these results, we conclude that declarative knowledge representations can be successfully used to produce imperative back-end code with non-trivial logic.

Keywords: low-code, model-driven web engineering, knowledge representation, ontologies, code generation.

1. 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 nine years ago [47], the term was used in the industry to label cloud-based development platforms that use visual notations to reduce the need for traditional

* Extended version of the paper "Beyond Low-Code Development: Marrying Requirements Models and Knowledge Representations" published at the FedCSIS'22 conference

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 [12] or as overlapping [50] with the Model-Driven Software Development (MDSD, MDD), where it concentrates on automatic generation of data-rich web/mobile applications from visual specifications (models).

A typical LCSD platform offers a visual language that enables the definition of user interface elements and application logic (navigation through the UI, handling user interactions). The language typically includes a visual notation to define the data model of the problem domain under consideration. These elements allow to generate and deploy automatically, much of a web/mobile application code. However, some non-standard data processing necessitates manual coding of appropriate APIs and services. In this article, we present an attempt to go beyond this and further reduce (or even eliminate) the need for traditional coding in software development.

The ultimate target of our research is a true general-purpose no-code development platform [67]. By this, we mean an environment that extends the low-code approach with ways to specify (business) domain logic and generate even complex data processing services. These services would integrate with the code generated from the low-code parts and thus enable creating a software system without programming in a traditional sense. What is essential, such a no-code approach would not be limited to a specific problem domain area.

When searching for candidate technology paths to fulfil the above vision, we can make certain observations. Specifically, low-code approaches use high-level, domain-agnostic abstractions familiar to domain experts (or software users). We can note that such abstractions can serve the same function as detailed software requirements in a typical software development project. Thus, we can expect to use specific requirements artefacts as the actual low-code source specifications. This way, the abstractions used in LCSD would be compatible with the abstractions used in traditional software processes.

So far, no research explicitly addresses the use of requirements artefacts in low-code approaches. However, we can refer to relevant research on Model-Driven Web Engineering (MDWE) that can be seen as a predecessor and a synonym for LCSD. A survey by Valderas and Pelechano [62] shows that most MDWE methods use some form of "classical" use case models [23] to specify functional requirements for web/mobile applications. They are usually supported by constrained language scenarios and/or activity diagrams to define detailed navigation through the user interface. Moreover, several approaches use some form of a data dictionary to define the problem domain. Based on this, certain automated transformations to other artefacts were proposed [28]. Nevertheless, most model-driven approaches cannot generate code directly from requirements.

One of the sparse systems that offer such capabilities is the ReDSeeDS[1] platform [57] which can be seen as a representative of requirements-based MDD. 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) [25, 39]. Specifications expressed in RSL allow for the generation of fully functional UI and application logic code for data-rich web applications. However,

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

based on RSL alone, one cannot generate data processing (domain logic) code beyond simple CRUD, and data persistence operations [68]. Thus, similarly to other low-code platforms, more complex data processing has to be coded manually.

Considering the above, 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, as they seem to share common goals. These approaches enable the representation of domain knowledge, independently of any technology and any particular problem domain. Ontologies offer means to create a shared, reusable vocabulary composed of concepts, rules and relationships between these concepts [16]. They constitute a common ground for communicating knowledge by domain experts and processing this knowledge by machines. For this purpose, they represent knowledge in a declarative way and are well adapted to conduct various reasoning tasks.

In this research, we investigate how the features of ontology-based domain logic representations 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). The new language draws several of its constructs from ontology-based knowledge representation approaches [53]. 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 an entirely 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.

The results presented in this article extend the results presented at the FedCSIS conference [51]. The extension includes a significant expansion of the presentation of the research context in the introduction, related work and RSL sections. We have also significantly extended the language examples and included the full list of translational rules with additional examples. Finally, we have added an additional validation approach which assesses complexity of RSL-DL code versus traditional code.

The structure of this article consists of nine sections. Section 2 describes the research context of our work, including overview of research on the low-code paradigm, visual languages and ontologies. Section 3 gives an overview of RSL which was the starting point for our research. Sections 4 and 5 define RSL-DL – its syntax and semantics. Section 6 includes the most important details of the language's transformation engine implementation. Sections 7 and 8 provide evaluation of the language through two case studies and three validation analyses. Section 9 concludes with a summary and discussion on future work. In addition, we also provide a Supplement (to be found in our github page[2]) that contains all the details that were omitted in the article for brevity.

2. Motivation and related work

The term "low-code" was probably first used in a Forrester report [47] only in 2014. Despite this relatively recent emergence, predictions for LCSD are promising – in 2019,

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

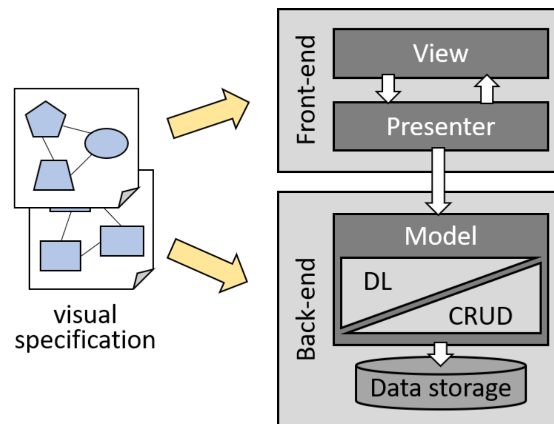

Fig. 1. Application generation in LCSD/MDWE

Gartner predicted that 65% of software in 2024 will be built using some form of low-code environment [63]. Still, there is not much data on the effectiveness of low-code approaches compared to more traditional ones. However, recent research by Bexiga et al. [7] and Trigo et al. [61] present very promising conclusions based on industrial experience. It is argued that this can be caused by a much higher involvement of design experts (e.g. UI/UX designers without programming skills) in software development. Yet, this does not eliminate the need for professional software developers' involvement and is not limited to simple applications produced by "citizen developers" [48]. Current studies report numerous industry-grade low-code platforms used extensively by professionals [52].

Sparse research results on LCSD can be supported by previous and current research on MDWE [29, 30] which can be seen as a predecessor and now a synonym for LCSD. A study by Wakil and Jawani [66] shows that research on MDWE is already quite broad and mature. Several studies show very significant productivity gains [33, 43, 14] and better maintainability [34]. It includes various controlled experiments and comparisons of complexity between high-level visual models and equivalent code. 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) [8]. Other examples – in the LCSD domain – are the Business Process Technology (BPT) language [21] and the Mendix notation [20].

The information scope of these LCSD/MDWE languages allows the associated development environments to generate significant portions of the target system's logic and the user interface. Figure 1 shows a typical approach to code generation found in such systems. It presents one of the popular architectural patterns, namely the Model-View-Presenter (MVP) [45]. The View (presentation) and Presenter (application logic) layers form the front-end of a system and can usually be fully generated from an LCSD/MDWE visual specification. The role of a code generation engine is to "inject" technology-specific elements in addition to the interaction flow and UI layouts already contained in the source visual specifications.

Still, LCSD/MDWE systems have limited capabilities regarding the generation of the domain/business logic code, or more broadly – the system's back-end (see again Figure 1). Current LCSD/MDWE languages can support generation of code for elementary CRUD (Create-Read-Update-Delete) operations [5, 49]. 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. [22] and Brito et al. [9]).

Our research is in line with the work by Atkinson et al. [4] 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 [19] 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 [58]. Appropriate works include more general discussions on introducing ontology-aware transformations [3, 42] and comprehensive formalised propositions on meta-models for ontology languages [44, 15]. An example of such a language is CoCoViLa by Haav, and Ojamma [18]. 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 [13] that use notations like JBoss Rules [11] or SBVR [40]. 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 (cf. Ontology-Driven Software Development [2, 17]). Stevenson and Dibson [59] propose a tooling framework for generating Java code from OWL specifications. Another example is work by Völkel and Sure [64] in which Java-based APIs are generated directly from ontologies expressed in RDF Schema. Especially relevant in the context of the low-code paradigm is the work of Krouwel et al. [32] who generate low-code Mendix applications from enterprise ontologies. Finally, our approach can be compared to that of Jbara et al. [24], and Vuorimaa et al. [65]. Both those solutions utilize declarative languages and have reported to have a positive impact on usability, especially for inexperienced ("citizen") developers and domain experts. However, our approach strives to provide more advanced code generation capabilities.

3. 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

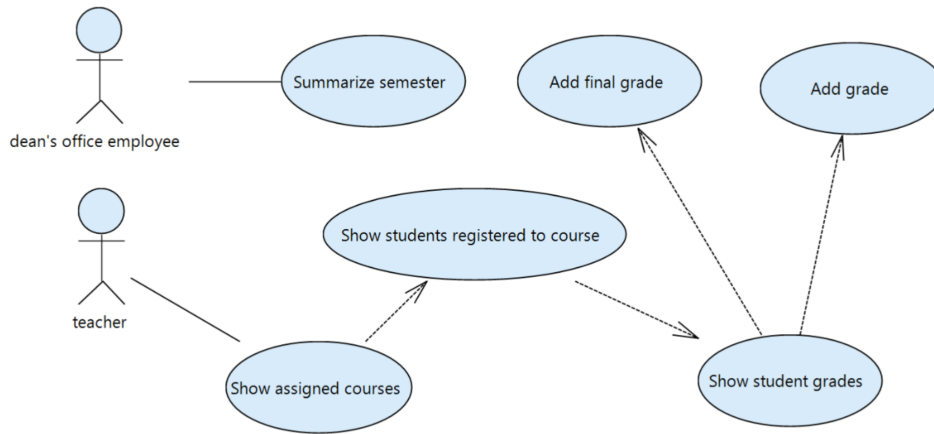

Fig. 2. Example use case model

between these two languages, some aspects of RSL-DL were strongly influenced by RSL. Here, we will present selected RSL constructs that will be most relevant in this context.

Figure 2 presents a simple use case model of a course management system that will serve as a frame for illustrating RSL. An important element worth explaining are the relationships between use cases. RSL has abandoned the ambiguous "extend" and "include" relationships of UML [6]. Instead, it has introduced the "invoke" relationship with procedure call semantics (see, e.g. [56], sec. 2.4.1). A detailed discussion of these relationships is out of the scope of this article. However, we will use an important feature of "invocation" which is parameter passing between use cases.

Figure 3 shows example scenarios of two of the use cases from Figure 2. Note that the "Add final grade" use case can be invoked from other use cases (here: "Show student grades"), which should pass two objects as parameters ("student" and "course"). These two parameters are defined in the "precondition" section of the respective scenario. The other part defines the application logic and contains a sequence of simple subject-verb-object sentences of various types. The most important of them from the point of view of this article are the "Query" type sentences. These sentences define actions of the system that are performed on certain data elements.

According to RSL semantics rules [55], we can transform scenarios into code. Figure 4 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 "Teacher selects save final grade" is translated into the "saveFinalGradeTriggered" 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 instance, the sentence "System computes weighted average grade data" is transformed into a call to the "computeWeightedAverageGradeData" service. UI presentation sentences are translated into calls to the View layer. For instance, the sentence "System shows semester summarized message" is transformed into a call to the "showSemesterSummarizedMessage" method. A more detailed

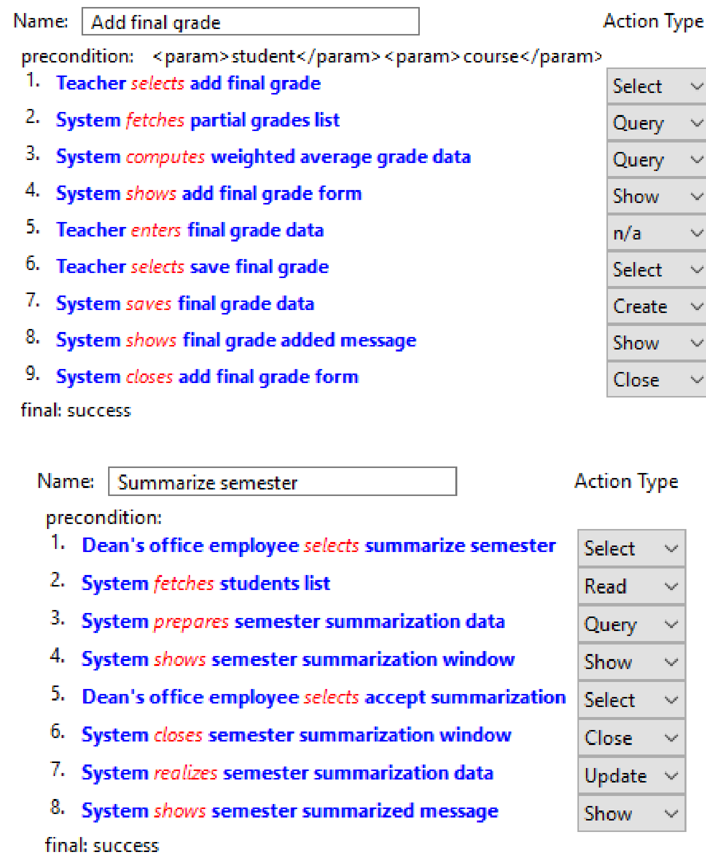

Fig. 3. Example scenarios

discussion of the rules and generated code, including code of the View layer, is presented elsewhere [56].

The relevant parts of the back-end service code generated from the RSL scenario is presented in Figure 5. 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. Note that other methods (not relevant to further presentation of RSL-DL) were omitted in Figure 5.

4. 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. Figure 6 presents an elementary example of concrete syntactic elements of the language. It contains definitions of four

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

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

Fig. 4. Presenter code generated for the use case scenarios

"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 Figure 6. 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 Figure 6) 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 Figure 7 using the MOF notation [41]. 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.

```
public class ServiceImpl implements IService {
    //...
    PartialGradeListDTO readPartialGradeList
        (Long inputStudentID, Long inputCourseID)
    {}

    WeightedAverageGradeDataDTO
        computeWeightedAverageGradeData
            (List<PartialGradeListItemDTO> partialGradeListDTO,
                    Long inputStudentID, Long inputCourseID)
    {}

    List<SemesterSummarizationDataItemDTO>
        preparesSemesterSummarizationData
            (List<StudentListItemDTO> studentListDTO)
    {}

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

Fig. 5. Back-end access code generated for the two example scenarios

Besides notions, there is a special kind of relationship participants – primitives ("DL-Primitive" 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

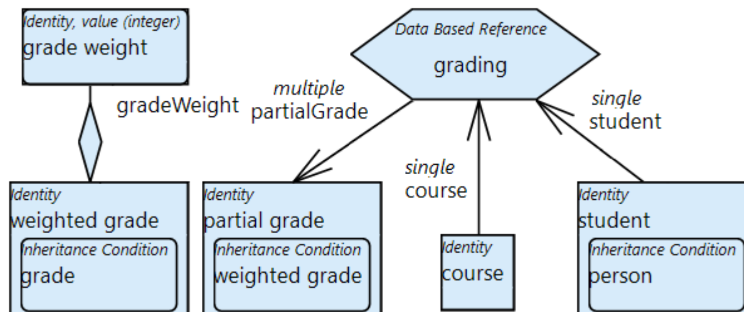

Fig. 6. RSL-DL concrete syntax example

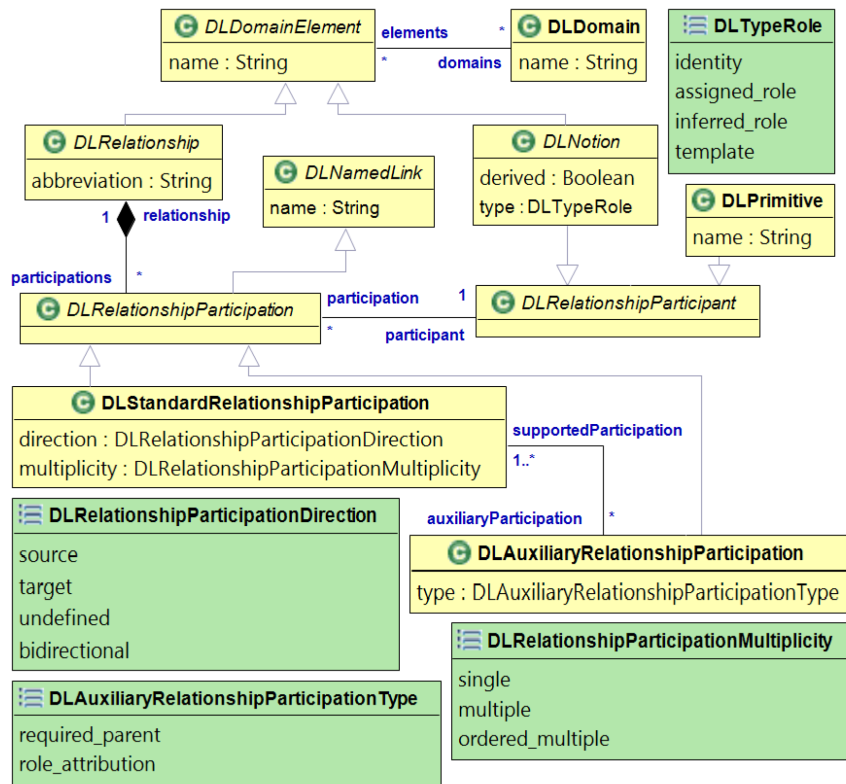

Fig. 7. Meta-model fragments for core RSL-DL elements

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 Figure 8. 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 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.

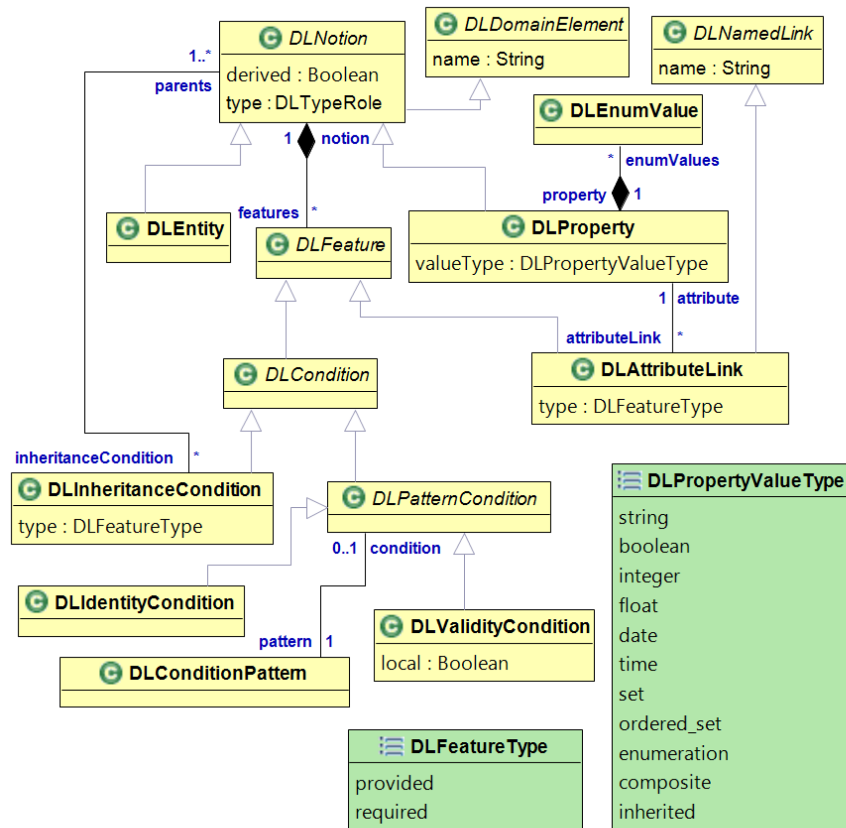

Fig. 8. Meta-model fragments for notions

Figure 9 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 [24, 65].

Figures 8 and 9 contain two additional elements: "DLTransitionPattern" and "DLConditionPattern". Each instance of these two meta-classes contains a string with a textual condition formula, a specific condition type and optionally – the condition's subject link

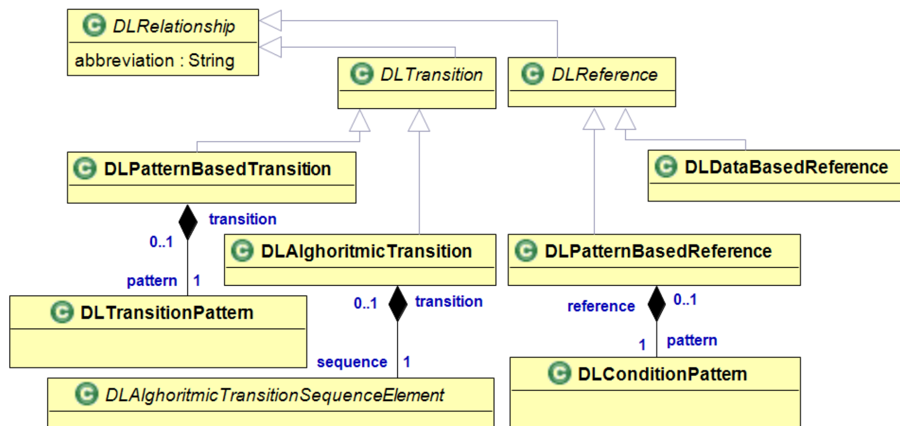

Fig. 9. Meta-model details for relationships

(see the detailed metamodel in the Supplement). The syntax of the condition is expressed in a language based on the notation used in the Symja library [31]. Some examples of several types of these patterns are given in section 7 in descriptions of the generated code.

5. Translational semantics rules

Of the many approaches to define semantics for RSL-DL (treated as a programming language [54]) 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 [26].

Full specification of semantics for RSL-DL consists of 16 translational rules (all the details can be accessed in the Supplement). The first ten rules define the generation 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. We summarise the responsibilities of specific rules in the following list.

- Rule 1 – generate classes from notions, except for those (“simple”) properties that do not contain other properties as attributes.
- Rule 2 – add inheritance dependencies between classes based on inheritance conditions.
- Rule 3 – generate a properly typed value field within a class generated from a “complex” property (a property that contains other properties as attributes).
- Rule 4 – generate fields derived from attribute links within notion-based classes.

- Rule 5 – generate appropriate constructors for notion-based classes, taking into account inheritance conditions and attribute links.
- Rule 6 – generate validation methods in notion-based classes, derived from validity conditions contained within the respective notions.
- Rule 7 – generate eligibility-check methods in notion-based classes, derived from identity conditions within the respective notions, taking into account inheritance conditions.
- Rule 8 – generate RUD (Read-Update-Delete) methods in notion-based classes.
- Rule 9 – generate classes from relationships and static methods within these classes; the generated methods retrieve objects of one type (cf. target relationship participant) derived from objects of other types (cf. source relationship participants).
- Rule 10 – generate additional methods in the above-generated classes; the methods check for the existence of particular relationships between objects potentially participating in a given relationship.
- Rule 11 – generate methods that respond to queries formulated within the application logic code; this rule allows to integrate RSL-DL-based code with code generated from RSL.
- Rule 12 – generate methods that compute values for attribute links marked as derived.
- Rule 13 – generate code that checks conditions based on condition patterns; this code is used in various other parts of the generated code.
- Rule 14 – generate code that filters objects based on condition patterns; used as in Rule 13.
- Rule 15 – generate code derived from transition patterns; used as in Rule 13.
- Rule 16 – generate auxiliary code that accesses full sets of objects of a given type or values corresponding to primitives; used as in Rule 13.

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 Figures 10, 12 and 13. 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, including the MOLA handbook [1] and a tutorial can be found on the MOLA website[3].

For brevity, we will limit our presentation of rule formalisation to two selected rules – 10 and 11. Figure 10 presents formalisation for the first of them. As the above list indicates, this particular rule defines the creation of methods (see: "check : Operation") that

[3] http://mola.mii.lu.lv/

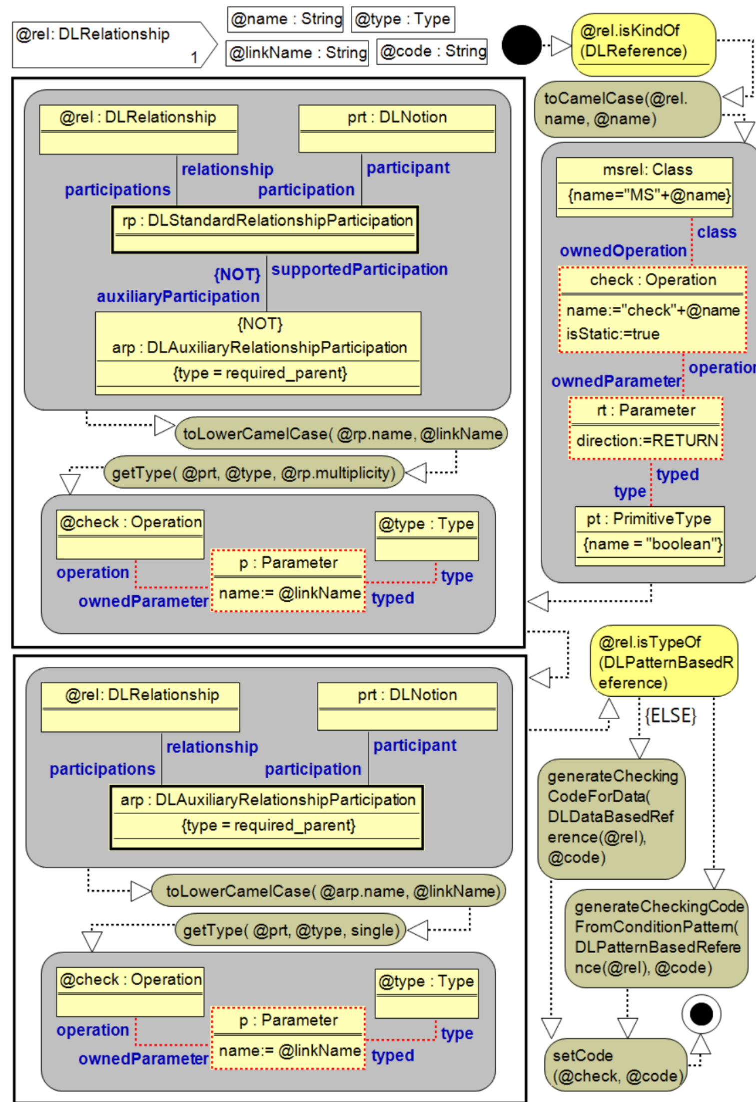

Fig. 10. Formalization of transformation rule 10

check the participation of objects in relationships. The generated method code returns a logical value (see: "rt : Parameter") and has parameters (see: "p : Parameter") generated through iterating over all the participants assigned to the relationship at hand. Since auxiliary participations need to be processed somewhat differently, the formalisation contains two such iterations. Additionally, appropriate procedures generating method code based on patterns ("generateCheckingCodeForData", "generateCheckingCodeFromCondition-Pattern") are invoked at the end. These procedures constitute formalisations of other rules (see Rules 13–16).

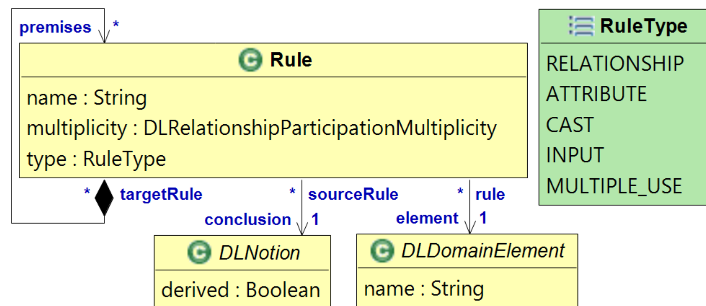

Fig. 11. Inference rule meta-model

To implement Rule 11, 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 Figure 3). 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 Figure 11. 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 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 Figure 12. 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 parameters that correspond to the rule's premises.

The actual formalisation of Rule 11 that uses the above algorithm is presented in Figure 13. 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.

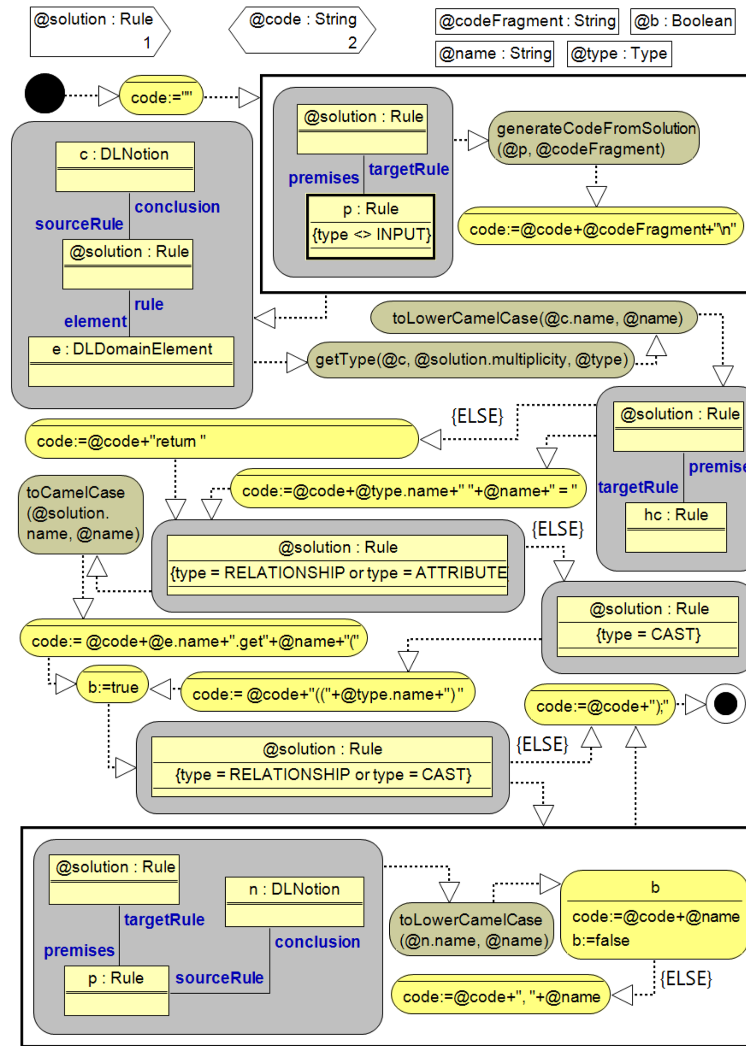

Fig. 12. Algorithm for generating the method body in rule 11

6. Implementation

The presented syntax and semantics of RSL-DL were implemented as an extension to the ReDSeeDS tool. We have constructed a dedicated RSL-DL editor using typical approaches to the construction of visual languages defined with metamodels. This includes the usage of Eclipse Modeling Framework (EMF) with Graphical Modeling Framework (GMF, currently superseded by the Sirius framework) for the implementation of the model repository and the graphical editor. This was then appended with a code generation component.

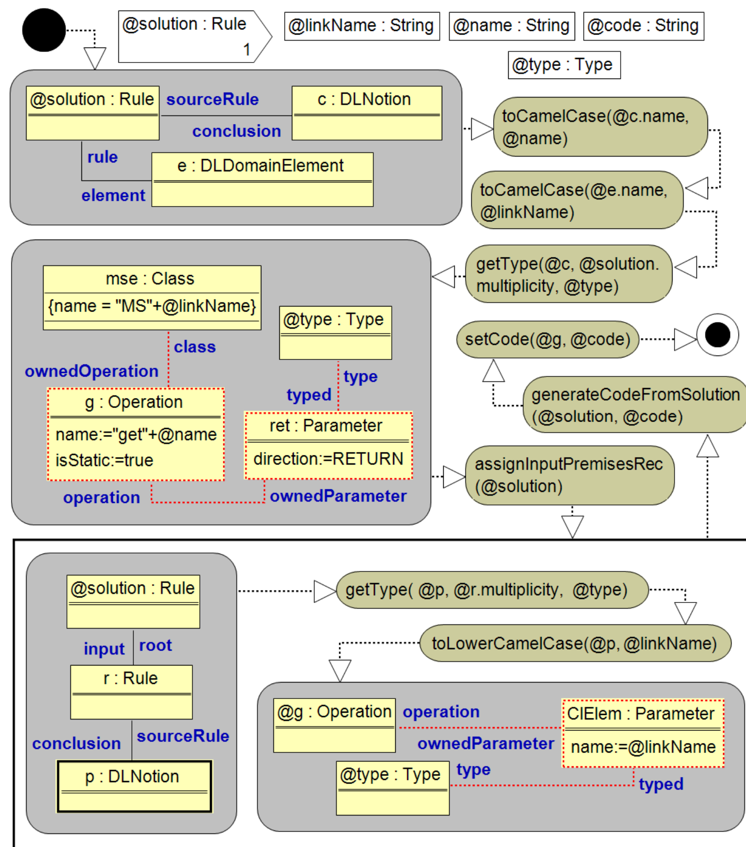

Fig. 13. Formalization of transformation rule 11

Figure 14 presents an overview of the code generation process. The inputs to the process are the RSL and RSL-DL models. The RSL model is used to generate front-end code, as explained in Section 3 [56] ("RSL-to-code transformation"). Scenario sentences from this model (typed as "Query") are also used to determine queries that form input to the inference engine. The inference engine takes these queries and applies them to the RSL-DL model. This results in a set of rules that will be used to generate domain code ("Rules necessary to implement domain operations"). These rules are organised into a tree structure that reflects full inference paths with references to appropriate domain elements from the RSL-DL model (see Figure 11).

The next step is to use a symbolic computation library to transform formulas contained in the rule tree. The goal is to obtain the formulas in a form suitable for the target logic (according to the queries). These transformed rules and the original RSL-DL model form inputs to the final code generation engine ("RSL-DL-to-code transformation"). This engine uses the rules presented in section 5 to produce the back-end code. This is joined with the front-end code that results in the final code for the system.

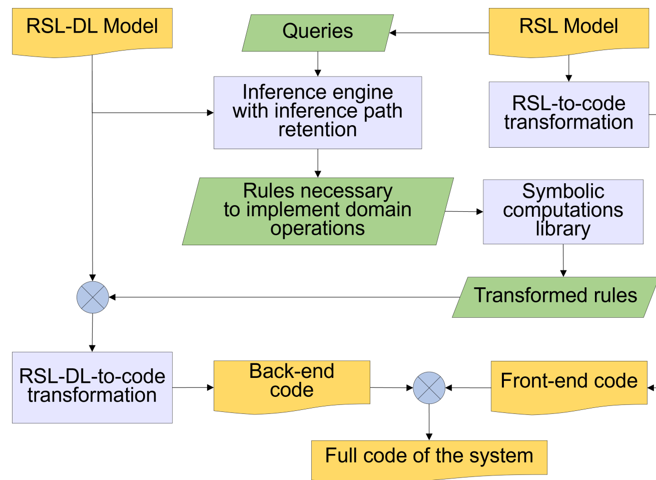

Fig. 14. Overview of the RSL-DL code generation process

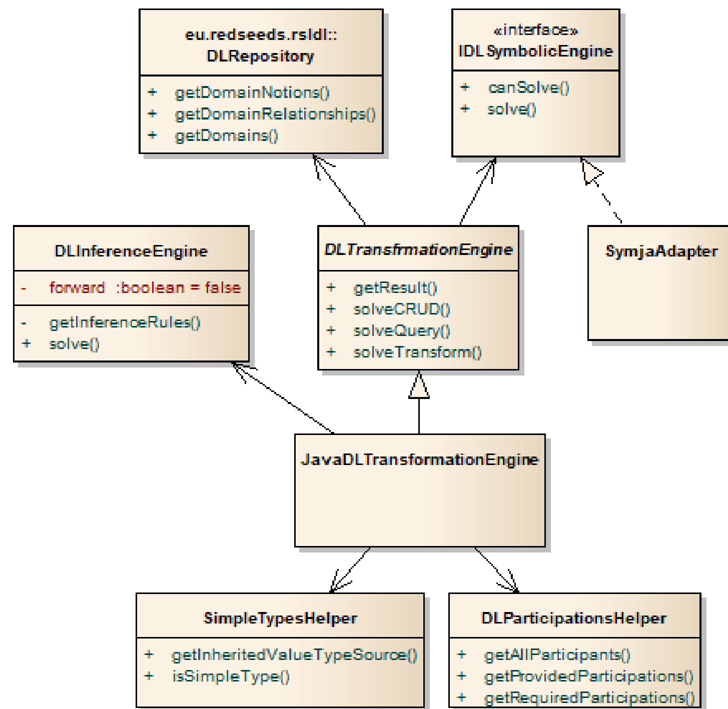

Fig. 15. Overview of the RSL-DL code generation module

The above described code generation component was implemented in Java as a plug-in to the ReDSeeDS tool. Its overall structure is presented in Figure 15 (we omit all the

technical details related to creating Eclipse plug-ins). The main element used to implement RSL-DL code generation is the "DLTransformationEngine" class. It defines the general scheme that the transformation should follow and serves as a basis for implementing transformation classes for specific target languages. As an example, we have implemented the "JavaDLTransformationEngine" class that generates Java code. Such code generators have access to model elements trough the "DLRepository" class, which provides instances of model elements based on EMF repository classes.

Access to symbolic computations is provided through the "IDLSymbolicEngine" interface. This interface contains operations for converting formulas contained in model elements (and inference rules). It is implemented using the SymJa symbolic computation library [31]. Transformation implementations should make use of three more provided classes: "DLInferenceEngine", "SimpleTypesHelper" and "DLParticipationsHelper". The "DLInferenceEngine" class implements the inference mechanisms described above. In response to queries, it provides the inference rule trees. The "SimpleTypesHelper" class helps in determining whether particular concepts should be represented in code as generated classes or built-in types. The "DLParticipationsHelper" class allows for more convenient handling of relationship participations during code generation. Full source code of the implementation can be accessed from the ReDSeeDS repository.

7. Case study

This section will present selected fragments of two more extensive case studies that illustrate several important uses of RSL-DL. The first case study refers to the functional requirements specification presented in Section 3. The second case study introduces a new problem domain (banking) and gives more insight into the inference engine and domain knowledge reuse.

7.1. Course Management System case

The RSL part of this case study was presented in Section 3. Thus, 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 3.

Figure 16a 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 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.

Figure 16b involves constructs for finding elements consistent with specific conditions. It defines the "academic year" as having the beginning and the end date. It also shows the "current academic year" relationship that enables the filtering of academic years

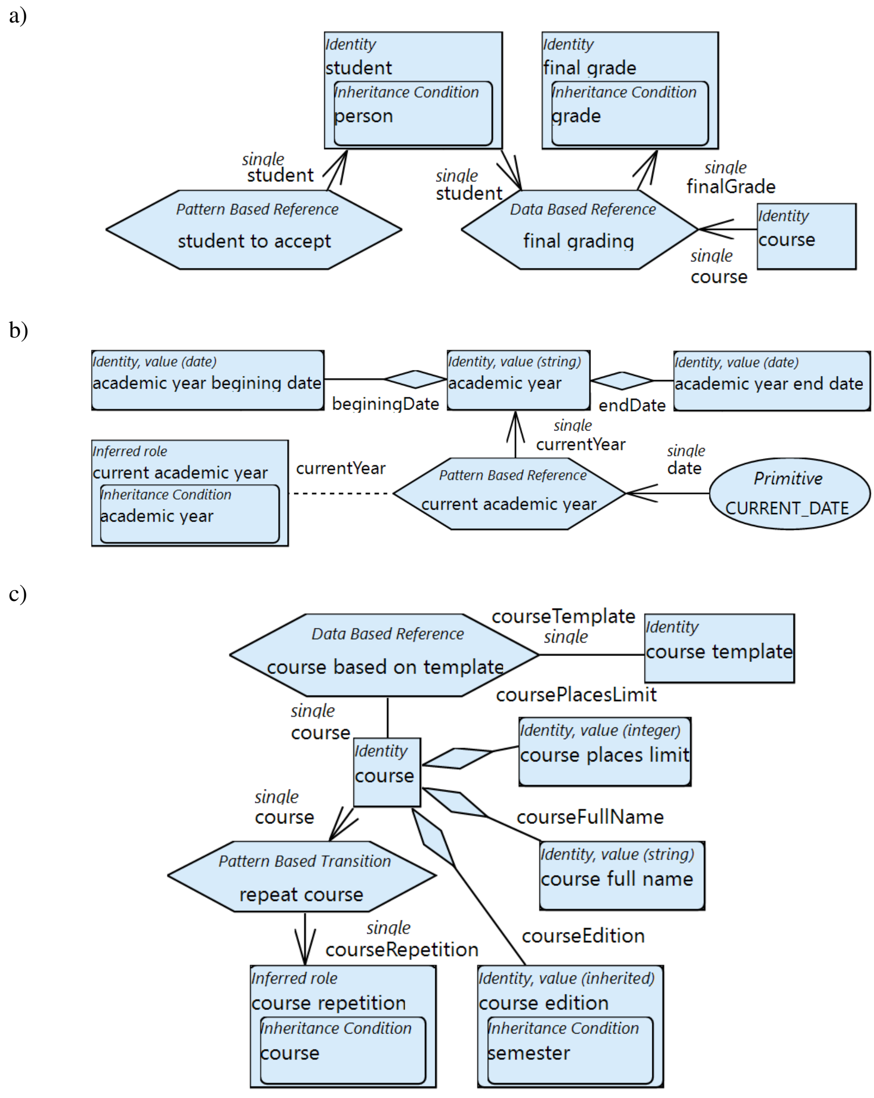

Fig. 16. RSL-DL model defining eligibility of students to be registered for the next semester (a), knowledge about academic years (b), and knowledge about course editions (c)

based on some condition. This condition (not shown in the figure) states that the current date should be between the beginning and the end date of the requested academic year. Additionally, the model distinguishes the “current academic year” notion. This notion is linked with the above relationship through an auxiliary relationship participation denoted with a dashed line. This means that whenever an inference is made based on the above relationship, to obtain an “academic year”, then this year can be treated as the “current academic year”. It allows for more convenient usage of conditions relating to academic years in other parts of the model. It is also worth noting that the multiplicity of the partic-

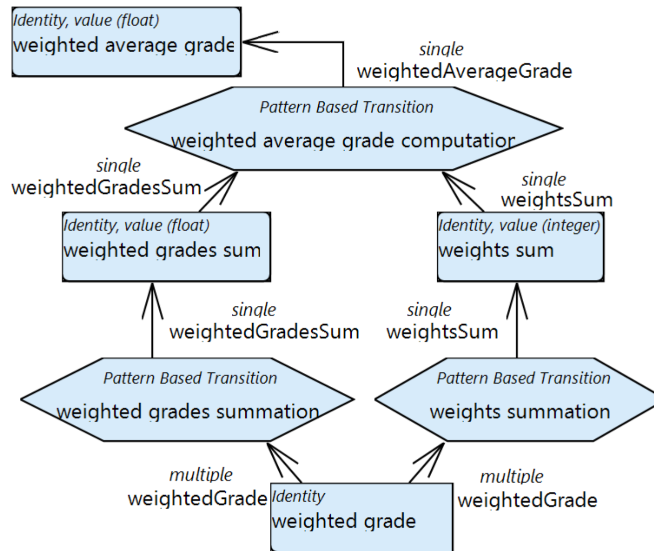

Fig. 17. RSL-DL model containing knowledge about computing of weighted average grades

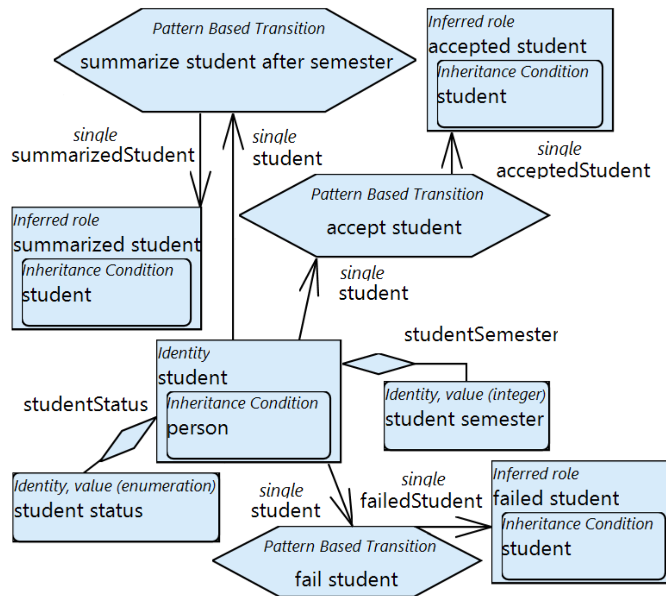

Fig. 18. RSL-DL model containing rules for promoting students to the next level of studies

ipation pointing to the academic year is denoted as single. Thus, there should be only one current academic year for the given “CURRENT_DATE”. Considering that the “CURRENT_DATE” is specified as a primitive, there should be only one current academic year at any given moment.

```
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. 19. Code for checking students' registration eligibility

Figure 16c involves constructs resulting in creating complex objects. It focuses on the "repeat course" relationship that allows creating another edition of a course for the current semester, based on one of the previous editions. The "previous" edition can be simply any course represented in the model as the participation leading from the "course" notion. The "new" course is represented as the "course repetition" notion, which also inherits from the "course" notion. The figure also shows additional information about courses. For example, each course is based on a "course template" through the "course based on template" relationship. Note that the direction of relationship participations indicate possible direction of inference and can be used to optimise the inference process.

Figure 17 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 "weights 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 Figure 18. The situation here is somewhat similar to the previous one. The whole modification is handled by the "summarize student after semester" transition, which requires two other transitions: "accept student", and "fail student". In contrast to the previous case, these transitions will be used interchangeably, depending on the fulfilment of a condition. This condition refers to the "student to accept" relationship presented in Figure 16a. If the student meets the "student to accept" relationship, the "accept student" transition is invoked, while in the opposite case, the "fail 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

```
public class MSCurrentAcademicYear {

    public static boolean checkCurrentAcademicYear
        (IMAcademicYear academicYear){
            LocalDate date = LocalDate.now();
            return academicYear.getBeginingDate().isBefore(date)
                && academicYear.getEndDate().isAfter(date);
        }

    public static IMCurrentAcademicYear getCurrentYear()
            List<IMAcademicYear> academicYears
                = MSAcademicYear.getAcademicYears();
            for (IMAcademicYear $_iter:academicYears)
                if (checkCurrentAcademicYear($_iter))
                    return new MCurrentAcademicYear($_iter);
        }

}
```

Fig. 20. Code for determining the current academic year

present some key fragments of code generated from the above-presented excerpts of the dean office model.

Figure 19 shows code generated on the basis of the model from Figure 16. 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" 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 Figure 16). 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 Figure 19. 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).

Figure 20 shows code generated on the basis of the model from Figure 16b. As in the previous code fragment, the generated supportive class contains two methods. The

```
public class MSRepeatCourse {

    public static List<IMCourseRepetition>
        getCourseRepetitions(List<IMCourse> courses){
            List<IMCourseRepetition> result
                = new ArrayList<IMCourseRepetition>();
            for (IMCourse $_iter:courses)
                result.add(getCourseRepetition($_iter));
            return result;
        }

    public static IMCourseRepetition
        getCourseRepetition(IMCourse course){
            IMCourse courseRepetition = new MCourse();
            courseRepetition.setCourseEdition(
                    MSNextSemester.getNext(
                            course.getCourseEdition()));
            courseRepetition.setCoursePlacesLimit(
                    course.getCoursePlacesLimit());
            courseRepetition.setCourseTemplate(
                    course.getCourseTemplate());
            return new MCourseRepetition(courseRepetition);
        }

}
```

Fig. 21. Code for creating new course editions

first one (“checkCurrentAcademicYear”) checks if the given academic year is the current one. It starts with code for obtaining the current date (transformation of the primitive “CURRENT_DATE”) according to rule no. 16 (generate auxiliary code). Following this, the method code returns a logical value computed based on the condition pattern in the “current academic year” relationship. The pattern formula (not shown in Figure 16b) has the value of “beginingDate(currentYear) < date && endDate(currentYear) > date” and is typed as “simple”. The second method in Figure 20 (“getCurrentYear”) uses the first to filter out academic years and return the current one. This method returns just a single object because the appropriate target participation is of the “single” type. It also casts the found year to the class corresponding to the “current academic year” notion (“MCurrentAcademicYear”). It is done based on rule no. 9 (generate classes and static methods from relationships), which determines the correct types within method signatures. In this case, the appropriate casting is done based on the “currentYear” auxiliary participation (“current academic year” notion participating in the “current academic year” relationship).

Figure 21 shows code generated on the basis of the model from Figure 16c. This time, the generated code creates objects representing new course editions based on previous ones. Nevertheless, again, the presented class (“MSRepeatCourse”) contains two methods generated according to rule no. 9 (generate classes and static methods from relationships). The first invokes the second one over a set of appropriate objects (list of courses). Contents of the second method (“getCourseRepetition”) are generated from rule no. 15 (generate code from transition patterns). It is translated from the transition pattern contained in the “repeat course” relationship. This pattern’s formula is “courseEdition(courseRepetition) == nextSemester (courseEdition(course)); coursePlacesLimit(courseRepetition) == coursePlacesLimit(course); template(courseRepetition) == template(course)”. It simply states

```
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. 22. Code for computing weighted average grades

that appropriate attributes of the repeated course should be the same as those in the base course or appropriately modified (here: the edition is incremented). This kind of transition pattern ("indirect") contains a set of "simple" transition pattern formulas that together allow for obtaining an object of the given type.

Figure 22 shows code generated on the basis of the model from Figure 17, using data structured according to Figure 6. 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 (generate methods for queries). 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 Figure 12). The first one of the overloaded methods is called from the second one. Its signature was generated according to rule no. 9 (generate classes and static methods from relationships). Its body was based on translating a transition pattern with the "simple" formula "weightedGradesSum/weightsSum" according to rule no. 15 (generate code from transition patterns).

Figure 23 shows code generated on the basis of the model from Figure 18. This time we can see only one method ("getSummarizedStudent") generated according to rule no. 9 (generate classes and static methods from relationships). 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 Figure 16. Thus, the current method calls the "checkStudentToAccept" method shown in Figure 19. Depending on its result, it calls one of two methods resulting from transforming the "accept student" and "fail student" relationships.

```
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. 23. Code for determining student's eligibility for promotion to the next semester

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

Fig. 24. Additional back-end access code

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 Figure 5). In our example this pertains to the method that corresponds to a non-CRUD operation. Appropriate additional code is presented in Figure 24. It contains a simple call to the operation presented in Figure 23. It is worth noting that such operations are optimised to use only required parameters.

7.2. Banking case study

In this study we consider a fragment of a banking application covering some operations regarding loans and deposits. Figure 25 contains four use cases that constitute the context for the example models and code. We will concentrate on the functionality of two use cases where their scenarios are presented in Figure 26. The first use case consists of two main parts. The first part allows the user to enter all the necessary data. In the second part,

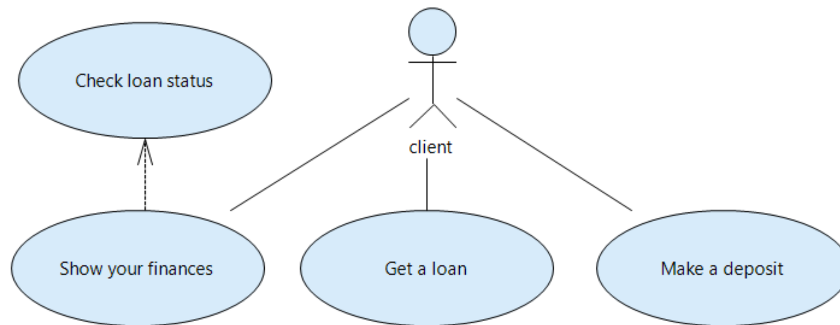

Fig. 25. Banking use case model fragment

the system shows information summarising the deposit for the user to confirm. Note that this gives two occasions to define domain logic. The first of them is to validate the deposit creation data entered by the user (sentence 5). The second one computes data to be shown in the deposit summary window (sentence 6).

Note that the result of the validation action (sentence 5) can lead to two alternate paths that depend on specific conditions ("=>cond" sentences). The first path is expressed with sentences 6-10. The second path is expressed by sentence 5.1.1 and the "rejoin" sentence. This forms application logic that is appropriately generated by the RSL-to-code transformation. The second use case contains simple logic to show loan status. Despite such simple scenario, the domain logic that computes such status can be quite complex.

Figure 27 shows the domain logic for validation, to be used in the first use case. The model contains elements that determine the possibility to create deposits based on given deposit parameters. Deposit parameters are concepts related to "deposit creation data", which is an RSL notion occurring in scenario sentences from Figure 26. Note that we have omitted this RSL model as quite trivial. The formula specifying data correctness is included in the validity condition assigned to the "deposit parameters" concept. It states that the funds on the selected source account should be equal or greater to the declared deposit amount. The model also presents the "source account" pattern-based reference which is used in the above formula. This is accompanied by all the attributes of the involved concepts ("deposit duration", "source account number", "account balance", "account number"). It is worth noting that many of these properties are specialised versions of more general terms used in representing more general dependencies further in the model.

Figure 28 shows three relationships related to various aspects of deposits – mainly general rules for calculating interests. Part a) defines that the "deposit compounding frequency" and the "annual nominal interest rate" depends on the deposit duration. This dependency is represented in the form of a data-based relationship. Part b) defines a formula for computing the final amount of a given deposit or loan which is obtained by using compound interest rules. Part c) defines the relation between the original sum of a given deposit or loan, its final amount and the total interest. The two latter formulas are represented as a pattern-based transitions.

Name: Make a deposit — Action Type

precondition:

1. Client selects make a deposit button — Select
2. System shows create deposit window — Show
3. Client enters deposit creation data — n/a
4. Client selects create deposit button — Select
5. System validates deposit creation data — Valida

=>cond: deposit creation data valid

6. System computes deposit summary data — Query
7. System shows deposit summary window — Show
8. Client selects confirm deposit button — Select
9. System creates deposit — Create
10. System shows deposit created message — Show

final: success

=>cond: deposit creation invalid

5.1.1 System shows invalid deposit message — Show

=>rejoin: Make a deposit — System shows create deposit window

Name: Check loan status — Action Type

precondition: <param>loan</param>

1. Client selects status button — Select
2. System computes loan status data — Query
3. System shows loan status window — Show

final: success

Fig. 26. Example scenarios for the banking case

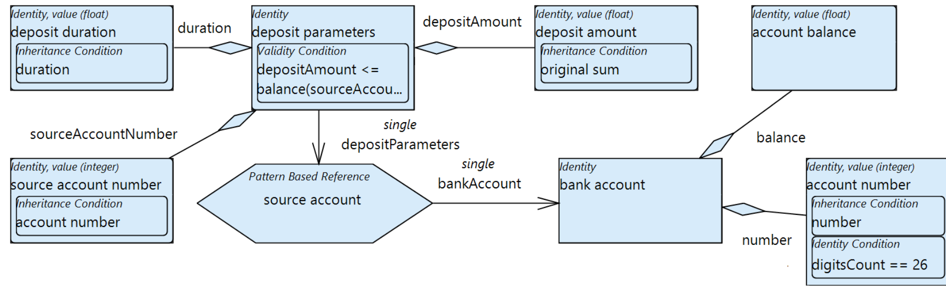

Fig. 27. RSL-DL model defining deposit creation data validation

Figure 29 shows two relationships related to various aspects of loans. Part a) defines the formula for calculating instalments of a loan. Part b) defines the formula for calculating the amount of debt left for a given loan if a certain number of its instalments were paid. Both of these cases are represented in the form of pattern-based transitions.

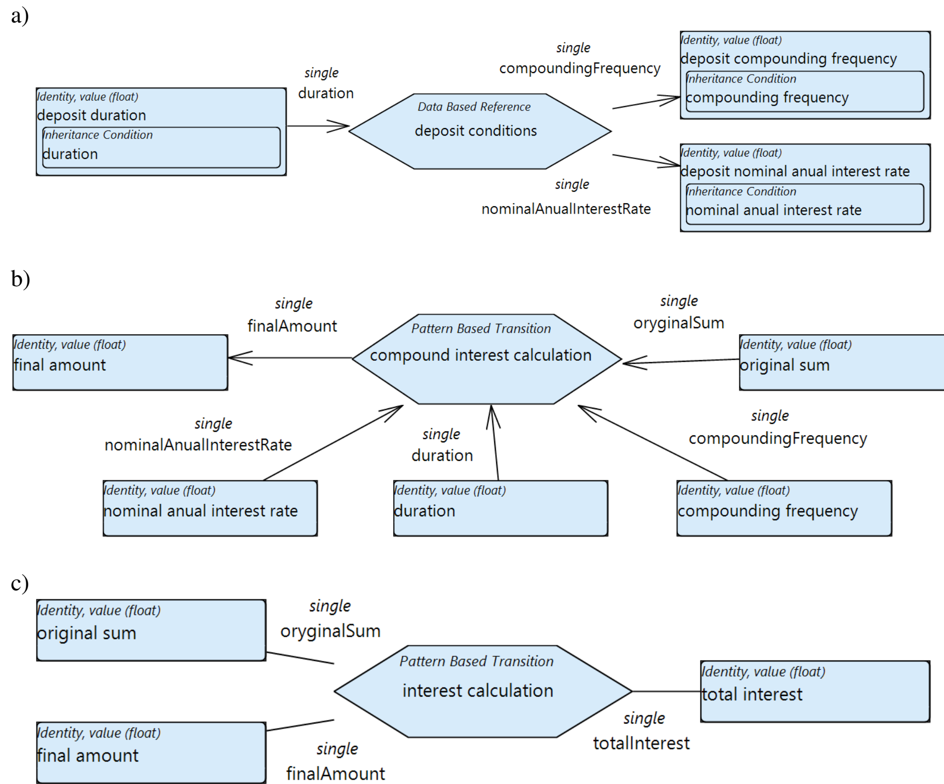

Fig. 28. RSL-DL model defining deposit conditions (a), compound interest calculation rules (b), and simple interest calculation rules (c)

Figure 30 shows code generated on the basis of the model from Figure 27. This code is a rather standard class with its attributes that were generated according to rules no. 1 (generate classes for notions) and no. 4 (generate attributes for attribute links) respectively. Note that the getters, setters, and constructors were omitted for brevity. In addition, we have the validation method, generated according to rule no. 6 (generate validation methods). The method checks validity of deposit parameters according to the validation condition formula from Figure 27. It uses the “MSSourceAccount” class generated from the “source account” reference according to rules no. 9 (generate classes and static methods from relationships) and 14 (generate object filtering code from condition patterns). This way, it is possible to obtain the bank account and its balance required by the formula.

Figure 31 shows code generated from the models in Figure 28. The goal is to obtain the “deposit summary” based on the provided “deposit parameters”. Note that the representation of “deposit summary” is not shown in the model for brevity. It is a concept related to the “deposit summary data” RSL notion found in the scenarios. It has five attributes: “deposit amount”, “deposit duration”, ”deposit nominal annual interest rate”, ”deposit compounding frequency” and “total interest”, where the last one is most com-

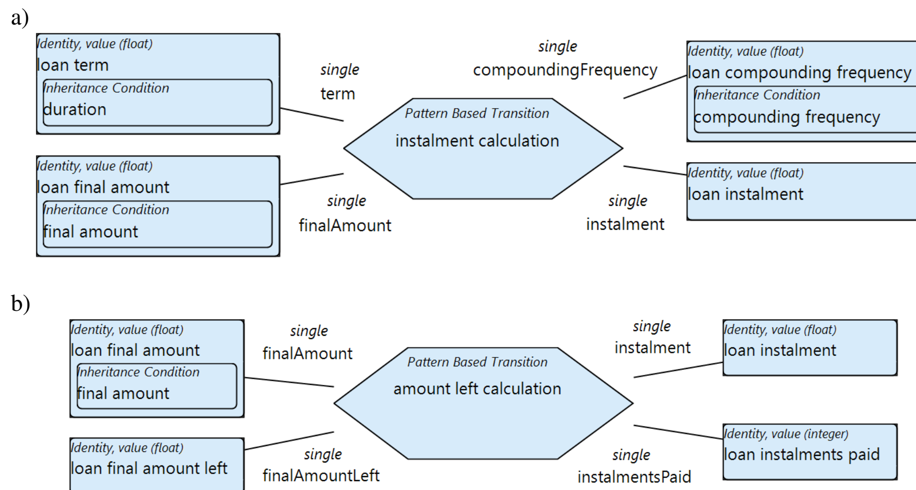

Fig. 29. RSL-DL model defining loan instalment calculation (a), and loan amount left calculation (b)

plex to obtain. This code was generated according to rule no. 11 (generate methods for queries). Its structure corresponds to the structure of the rule tree generated by the inference engine to allow for computing of these values.

The "getDepositSummary" method starts by obtaining the deposit sum and its duration, which are easily available as attributes of the "MDepositParameters" class (cf. Figures 27 and 30). Then, the "nominal annual interest rate" and the "compounding frequency" are obtained using appropriate methods of the "MSDepositCondition" class. These methods were generated according to rule no. 9 (generate classes and static methods from relationships) from the "deposit conditions" data-based reference (see Figure 28a). Finally, appropriate methods of the "MSCompoundInterestCalculation" and the "MSInterestCalculation" class are used to obtain the final amount and the total interest for a

```
public MDepositParameters implements IMDepositParameters {

    private double depositAmount;
    private double depositDuration;
    private int sourceAccountNumber;

    (...)

    public boolean validate(){
        return depositAmount <= MSSourceAccount
            .getBankAccount(this).getBalance();
    }

}
```

Fig. 30. Code for deposit creation data validation

```
class MSDepositSummary {

    public static IMDepositSummary getDepositSummary
            (IMDepositParameters depositParameters){
        double depositAmount = depositParameters.getDepositAmount();
        double depositDuration = depositParameters.getDuration();
        double depositNominalAnualInterestRate =
            MSDepositConditions.getNominalAnualInterestRate(depositDuration);
        double depositCompoundingFrequency =
            MSDepositConditions.getCompoundingFrequency(depositDuration);
        double finalAmount = MSCompoundInterestCalculation.getFinalAmount
            (depositAmount, depositNominalAnualInterestRate,
                    depositCompoundingFrequency, depositDuration);
        double totalInterest = MSInterestCalculation.getTotalInterest
            (finalAmount, depositAmount);
        return new MDepositSummary(depositAmount, depositDuration,
                depositNominalAnualInterestRate, depositCompoundingFrequency,
                totalInterest);
    }

}
```

Fig. 31. Code for deposit summary computation

```
class MSCompoundInterestCalculation {

    public static double getFinalAmount(double originalSum,
            double nominalAnualInterestRate,
            double compoundingFrequency, double duration){
        return originalSum*Math.pow(1+nominalAnualInterestRate/
                compoundingFrequency,
                compoundingFrequency*duration);
    }

}
```

Fig. 32. Code for compound interest calculation

deposit. Both classes are generated according to rules no. 9 and 15 (generate code from transition patterns) from the ”compound interest calculation” and the ”interest calculation” pattern-based transitions (Figure 28b and c). The last step is to return the “deposit summary” object, created from the previously obtained values by using the constructor generated according to rule no. 5 (generate constructors for notion-based classes).

The details of the “MSCompoundInterestCalculation” class used in the above code are shown in Figure 32. The actual code of its only method is derived from the formula “$originalSum * (1 + nominalAnualInterestRate/compoundingFrequency)\hat{}$ $(compoundingFrequency * duration)$” included in the respective pattern-based transition (see Figure 28b).

Figure 33 shows code generated from the model in Figure 29. Its goal is to obtain the “loan status” based on a provided object representing a loan. Note that the the representation of “loan status” is not shown in the model for brevity. It is a concept related to the “loan status data” RSL notion found in the scenarios. It specialises from the “loan” concept and has five additional attributes: “loan nominal annual interest rate”, “loan com-

```
class MSLoanStatus {

    public static IMLoanStatus getLoanStatus(IMLoan loan){
        double originalSum = loan.getLoanAmount();
        double loanDuration = loan.getDuration();
        double loanNominalAnualInterestRate =
            MSLoanConditions.getNominalAnualInterestRate(loanDuration);
        double loanCompoundingFrequency =
            MSLoanConditions.getCompoundingFrequency(loanDuration);
        double finalAmount = MSCompoundInterestCalculation.getFinalAmount
            (originalSum, loanNominalAnualInterestRate,
                    loanCompoundingFrequency, loanDuration);
        double totalInterest = MSInterestCalculation.getTotalInterest
                    (finalAmount, originalSum);
        double loanInstalment = MSInstalmentCalculation.getInstalment
            (finalAmount, loanCompoundingFrequency, loanDuration);
        int loanInstalmentsPaid = loan.getInstalmentsPaid();
        double loanFinalAmountLeft = MSAmountLeftCalculation.getFinalAmountLeft
            (finalAmount, loanInstalments, loanInstalmentsPaid);
        return new MLoanStatus(loan, loanNominalAnualInterestRate,
                loanCompoundingFrequency, loanFinalAmountLeft, loanInstalment,
                totalInterest);
    }

}
```

Fig. 33. Code for loan status computation

pounding frequency", "loan final amount left", "loan instalment" and "total interest". Most of these values overlap with these from the previous code fragment, thus majority of this code is very similar (applied to a loan instead of a deposit). There is however the need to compute two new values – the "loan instalment" and the "loan final amount". Both of these values are obtained by using the class and its methods generated according to rules no. 9 and 15 from the "instalment calculation" and the "amount left calculation" pattern-based transitions (see Figure 29). To obtain the second of these values, the number of loan instalments already paid is also needed, so its value is obtained beforehand from the appropriate attribute of the "MLoan" class.

8. Language validation and discussion

In the previous section, we presented the most interesting parts of code automatically generated from the RSL-DL domain specifications of two cases. This code constituted the back-end of the respective systems and was compatible with the front-end code generated from the RSL specifications. It has to be stressed that the generated code covered complete back-end logic, and almost no manual updates were necessary. The only significant update necessary in the current generator version was adding a database access code. We did not cover this element due to that it was already developed previously, including an appropriate automated transformation for the ReDSeeDS system [68]. All other issues are minor (e.g. handling of import statements) and should be easily solvable through further improvements to the transformation code.

It should be noted that currently, the two languages - RSL and RSL-DL are only partially integrated. In particular, RSL contains its own domain modelling language that is

focused on the front-end perspective. The presented transformations (RSL-to-code and RSL-DL-to-code) are compatible with respect to the structural composition of the generated code. However, these transformations have to be invoked separately and then manually integrated by adding small fragments of interfacing code. To obtain a truly no-code environment, we would first need to fully integrate the domain modelling part of RSL with the syntax and semantics of RSL-DL. Furthermore, we would need to construct a unified transformation that generates fully integrated front-end and back-end code. This should also include the generation of the database structure and data persistency code.

Proper validation of a full no-code system based on RSL and RSL-DL is thus future work. This would involve using prospective ReDSeeDS 2.0 in real-world projects where specifications in RSL+RSL-DL would remove the need to write traditional code completely. However, even partial validation of RSL-DL alone can give interesting insights on the usefulness and improvements that would need to be made to create such a system.

To initially validate RSL-DL, we have used primarily 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. Additionally, we have also conducted a brief complexity analysis of RSL-DL language constructs in comparison to more traditional ones.

8.1. 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 metamodelling. 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%. The above results show that even after a brief introduction, most of issues could be understood correctly, even by non-professionals (students). An important result of this study is the notion of which parts of the questionnaire caused most problems for the participants. 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 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.

8.2. 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 1.

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 (back-end) 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.

Table 1. Results of the operability study

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

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.

8.3. Analysis of complexity

An important and obvious aspect of RSL-DL design as a low-code language was to reduce complexity of code. To determine characteristics of RSL-DL in this aspect, we need to compare the complexity of RSL-DL models (including model queries) and equivalent 3GL back-end code. We can find several metrics of software (textual code) complexity in literature [27]. Moreover, relatively sparse complexity metrics exist for visual models (mainly formulated in UML) [36]. However, we could not find a generally accepted objective metric for comparing the complexity of textual code and visual models (cf. work by Masmali and Badreddin [37]). Considering this, to assess the potential reduction of complexity, we have applied a simple metric that compares quantities of syntactic constructs.

Our comparative procedure followed the following steps. First, we have manually developed appropriate domain logic code for the specified use cases and scenarios. This involved manual implementation of services, like IService in Figures 5 and 24. Second, after implementing the transformation from RSL-DL to code, we have specified the domain using RSL-DL notation. Finally, after generating the code, we compared it to that developed manually to ensure that they are functionally equivalent.

Quantitative comparison of syntactic constructs shows at least a 30% reduction in complexity for RSL-DL models. As an example, we can consider relevant model and code fragments for computing "weighted average grade" (compare Figures 6, 17 and 22). The model consists of 22 syntactic elements (3 relations, 7 notions, 7 participations, 2 attribute links and 3 formulas) to which we need to add 1 query. The equivalent code consists of 35 syntactic elements (5 classes, 2 fields, 4 methods, 2 constructors, 4 getter methods, 2 loops, and 16 other instructions). It has to be stressed that the above numbers pertain to

manually developed code and not that which was automatically generated. However, differences were minimal and mainly concerned with optimisation (manual code was more optimised and more compact).

The above comparison can be treated as indicative and necessitates more detailed experimentation and analysis, which we see as future work. Regardless of this, significant benefits can also be achieved due to the reusability capabilities of RSL-DL. Knowledge representations expressed in RSL-DL are declarative and independent of any particular context of usage. Thus they can be queried similarly to how ontologies can be queried for inferring various facts. However, RSL-DL queries result not in facts but ready imperative code adapted to the given usage (application logic). Therefore, the only cost of obtaining code adapted to new functional specification (e.g. new use cases and their scenarios) is the cost of formulating new queries.

9. Summary and future work

This work attempted at going beyond typical low-code approaches. We have applied declarative knowledge representations to define non-trivial logic that can be used to produce imperative back-end code in a standard programming language. This generated code can be interfaced with front-end code produced from low-code specifications that use formalised requirements models (use cases, scenarios). It thus can be noted that our approach leads to shifting most of the programming activities to the level typically used in requirements engineering. Programming becomes equivalent to specifying precise requirements 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” [35] and a step towards a “fifth generation language” as postulated by Thalheim and Jaakkola [60].

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 [10]). 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 RSL-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 [24, 65]). 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 [46]. 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. [38]. 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 [68].

Finally, we would like to address the fact that both RSL and RSL-DL were created as distinct new languages. Still, they follow typical notations found in requirements engineering, domain modelling and knowledge representation. It thus can be argued that an existing general-purpose language (e.g. a UML profile) could be used to express the same semantics. Both approaches – creating a new language vs. using a general-purpose language – can be seen as equivalent. However, creating a language from scratch allowed us to go beyond the "beaten path" and create visual syntax that would be hard to express using standard notations.

References

1. The MOLA Language Reference Manual Version 2.0 final (2007)
2. Aßmann, U., Ebert, J., Pan, J.Z., Staab, S., Zhao, Y.: Ontology-Driven Software Development. Springer Berlin Heidelberg, Berlin, Heidelberg (2013)
3. Aßmann, U., Zschaler, S., Wagner, G.: Ontologies, Meta-models, and the Model-Driven Paradigm, pp. 249–273. Springer (2006)
4. Atkinson, C., Gutheil, M., Kiko, K.: On the relationship of ontologies and models. In: Proc. 2nd Workshop on Meta-Modelling, WoMM 2006. pp. 47–60 (2006)
5. Basso, F.P., Pillat, R.M., Oliveira, T.C., Roos-Frantz, F., Frantz, R.Z.: Automated design of multi-layered web information systems. Journal of Systems and Software 117, 612–637 (2016)
6. van den Berg, K.G., Simons, A.J.H.: Control flow semantics of use cases in UML. Information and Software Technology 41(10), 651–659 (1999)
7. Bexiga, M., Garbatov, S., Seco, J.a.C.: Closing the gap between designers and developers in a low code ecosystem. In: 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings. pp. 1–10. MODELS '20, Association for Computing Machinery, New York, NY, USA (2020)
8. Brambilla, M., Fraternali, P.: Interaction flow modeling language: Model-driven UI engineering of web and mobile apps with IFML. Morgan Kaufmann (2014)
9. Brito, I.S., Barros, J.P., Gomes, L.: From requirements to code (Re2Code) – a model-based approach for controller implementation. In: 2016 IEEE 14th International Conference on Industrial Informatics (INDIN). pp. 1224–1230 (2016)
10. Brooks, F.P.: No silver bullet: Essence and accidents of software engineering. IEEE Computer 20(4), 10–19 (April 1987)
11. Browne, P.: JBoss Drools business rules. Packt Publishing Ltd (2009)
12. Cabot, J.: 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)
13. Chisholm, M.: How to build a business rules engine: extending application functionality through metadata engineering. Morgan Kaufmann (2004)
14. Fatolahi, A., Some, S.S., et al.: Assessing a model-driven web-application engineering approach. Journal of Software Engineering and Applications 7(05), 10p (2014)
15. Gašević, D., Djurić, D., Devedžić, V.: Model Driven Engineering and Ontology Development. Springer (2009)
16. Gruber, T.R.: A translation approach to portable ontology specifications. Knowledge Acquisition 5(2), 199–220 (1993)
17. Haav, H.M.: A comparative study of approaches of ontology driven software development. Informatica 29(3), 439–466 (2018)

18. Haav, H.M., Ojamaa, A.: Semi-automated integration of domain ontologies to DSL metamodels. International Journal of Intelligent Information and Database Systems 10(1/2), 94–116 (2017)
19. Henderson-Sellers, B.: Bridging metamodels and ontologies in software engineering. Journal of Systems and Software 84(2), 301–313 (2011)
20. Henkel, M., Stirna, J.: Pondering on the key functionality of model driven development tools: The case of Mendix. In: International Conference on Business Informatics Research. pp. 146–160. Springer (2010)
21. Henriques, H., Lourenço, H., Amaral, V., Goulão, M.: Improving the developer experience with a low-code process modelling language. In: 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems. pp. 200–210 (2018)
22. Hinchey, M.G., Rash, J.L., Rouff, C.A.: Requirements to design to code: Towards a fully formal approach to automatic code generation. Tech. rep., NASA (2005)
23. Jacobson, I., Christerson, M., Jonsson, P., Övergaard, G.: Object-Oriented Software Engineering: A Use Case Driven Approach. Addison-Wesley (1992)
24. Jbara, A., Bibliowicz, A., Wengrowicz, N., Levi, N., Dori, D.: Toward integrating systems engineering with software engineering through object-process programming. International Journal of Information Technology pp. 1–35 (2020)
25. Kaindl, H., Smialek, M., Wagner, P., et al.: Requirements specification language definition. Project Deliverable D2.4.2, ReDSeeDS Project (2009)
26. Kalnins, A., Barzdins, J., Celms, E.: Model transformation language MOLA. Lecture Notes in Computer Science 3599, 62–76 (2004)
27. Khan, A.A., Mahmood, A., Amralla, S.M., Mirza, T.H.: Comparison of software complexity metrics. International Journal of Computing and Network Technology 4(1), 19–26 (2016)
28. Koch, N., Kozuruba, S.: Requirements models as first class entities in model-driven web engineering. In: Current Trends in Web Engineering. pp. 158–169. Springer (2012)
29. Koch, N., Kraus, A.: Towards a common metamodel for the development of web applications. In: Web Engineering. pp. 497–506. Springer (2003)
30. Koch, N., Meliá-Beigbeder, S., Moreno-Vergara, N., Pelechano-Ferragud, V., Sánchez-Figueroa, F., Vara-Mesa, J.: Model-driven web engineering. Upgrade-Novática Journal (English and Spanish) 2, 40–45 (2008)
31. Kramer, A.: Symja library-Java symbolic math system (2010), https://github.com/axkr/symja_android_library, last accessed June 2023
32. Krouwel, M.R., Land, M.O.t., Proper, H.A.: Generating low-code applications from enterprise ontology. In: Barn, B.S., Sandkuhl, K. (eds.) The Practice of Enterprise Modeling. pp. 18–32. Springer International Publishing, Cham (2022)
33. Martínez, Y., Cachero, C., Meliá, S.: Evaluating the impact of a model-driven web engineering approach on the productivity and the satisfaction of software development teams. In: Web Engineering. pp. 223–237. Springer (2012)
34. Martínez, Y., Cachero, C., Meliá, S.: Empirical study on the maintainability of web applications: Model-driven engineering vs code-centric. Empirical Software Engineering 19(6), 1887–1920 (2014)
35. Martins, B.F.: The OntoOO-method: An ontology-driven conceptual modeling approach for evolving the oo-method. In: Advances in Conceptual Modeling. pp. 247–254 (2019)
36. Masmali, O., Badreddin, O.: Comprehensive model-driven complexity metrics for software systems. In: 20th International Conference on Software Quality, Reliability and Security Companion (QRS-C). pp. 674–675 (2020)
37. Masmali, O., Badreddin, O.: Towards a model-based fuzzy software quality metrics. In: MODELSWARD. pp. 139–148 (2020)
38. Mefteh, M., Bouassida, N., Ben-Abdallah, H.: Towards naturalistic programming: Mapping language-independent requirements to constrained language specifications. Science of Computer Programming 166, 89–119 (2018)

39. Nowakowski, W., Śmiałek, M., Ambroziewicz, A., Straszak, T.: Requirements-level language and tools for capturing software system essence. Computer Science and Information Systems 10(4), 1499–1524 (2013)
40. Object Management Group: Semantics of Business Vocabulary and Business Rules, version 1.4, formal/2017-05-05 (2017)
41. Object Management Group: Meta Object Facility (MOF) Core Specification, version 2.5.1, formal/2019-10-01 (2019)
42. Pahl, C.: Semantic model-driven architecting of service-based software systems. Information and Software Technology 49(8), 838–850 (2007)
43. Papotti, P.E., Do Prado, A.F., de Souza, W.L., Cirilo, C.E., Pires, L.F.: A quantitative analysis of model-driven code generation through software experimentation. In: International Conference on Advanced Information Systems Engineering. pp. 321–337 (2013)
44. Parreiras, F.S., Staab, S., Schenk, S., Winter, A.: Model driven specification of ontology translations. In: ER'08. Lecture Notes in Computer Science, vol. 5231, pp. 484–497 (2008)
45. Potel, M.: MVP: Model-View-Presenter the Taligent programming model for C++ and Java. Tech. rep., Taligent Inc. (1996)
46. Pulido-Prieto, O., Juárez-Martínez, U.: A survey of naturalistic programming technologies. ACM Comput. Surv. 50(5), 35p (2017)
47. Richardson, C., Rymer, J.R., Mines, C., Cullen, A., Whittaker, D.: New development platforms emerge for customer-facing applications (2014), Forrester report
48. Richardson, C., Rymer, J.R., Mines, C., Cullen, A., Whittaker, D.: Vendor landscape: The fractured, fertile terrainof low-code application platforms (2016), Forrester report
49. Rodriguez-Echeverria, R., Preciado, J.C., Rubio-Largo, A., Conejero, J.M., Prieto, A.E.: A pattern-based development approach for Interaction Flow Modeling Language. Scientific Programming 2019, 1–15 (apr 2019)
50. Ruscio, D.D., Kolovos, D., de Lara, J., Pierantonio, A., Tisi, M., Wimmer, M.: Low-code development and model-driven engineering: Two sides of the same coin? Software and Systems Modeling pp. 1–10 (jan 2022)
51. Rybiński, K., Śmiałek, M.: Beyond low-code development: Marrying requirements models and knowledge representations. In: 2022 17th Conference on Computer Science and Intelligence Systems (FedCSIS). pp. 919–928 (2022)
52. Sahay, A., Indamutsa, A., Di Ruscio, D., Pierantonio, A.: Supporting the understanding and comparison of low-code development platforms. In: 46th Euromicro Conference on Software Engineering and Advanced Applications. pp. 171–178 (2020)
53. Slimani, T.: Ontology development: A comparing study on tools, languages and formalisms. Indian Journal of Science and Technology 8(24), 1–12 (2015)
54. Slonneger, K., Kurtz, B.L.: Formal Syntax and Semantics of Programming Languages. Addison-Wesley (1995)
55. Śmiałek, M., Jarzebowski, N., Nowakowski, W.: Runtime semantics of use case stories. In: Visual Languages and Human-Centric Computing (VL/HCC), 2012 IEEE Symposium on. pp. 159–162. IEEE (2012)
56. Śmiałek, M., Nowakowski, W.: From Requirements to Java in a Snap: Model-Driven Requirements Engineering in Practice. Springer (2015)
57. Smialek, M., Straszak, T.: Facilitating transition from requirements to code with the ReDSeeDS tool. In: Requirements Engineering Conference (RE), 2012 20th IEEE International. pp. 321–322 (2012)
58. Soylu, A., De Causmaecker, P.: Merging model driven and ontology driven system development approaches: Pervasive computing perspective. In: 24th International Symposium on Computer and Information Sciences. pp. 730–735 (2009)
59. Stevenson, G., Dobson, S.: Sapphire: Generating java runtime artefacts from OWL ontologies. In: Progress in Pattern Recognition, Image Analysis, Computer Vision, and Applications, pp. 425–436. Springer (2011)

60. Thalheim, B., Jaakkola, H.: Model-based fifth generation programming. In: Information Modelling and Knowledge Bases XXXI, pp. 381–400. IOS Press (2020)
61. Trigo, A., Varajão, J., Almeida, M.: Low-code versus code-based software development: Which wins the productivity game? IT Professional 24(5), 61–68 (2022)
62. Valderas, P., Pelechano, V.: A survey of requirements specification in model-driven development of web applications. ACM Trans. Web 5(2), 1–51 (2011)
63. Vincent, P., Iijima, K., Driver, M., Wong, J., Natis, Y.: Magic quadrant for enterprise low-code application platforms. Gartner report (2019)
64. Völkel, M., Sure, Y.: RDFReactor-from ontologies to programmatic data access. In: Poster Proceedings of the Fourth International Semantic Web Conference (2005)
65. Vuorimaa, P., Laine, M., Litvinova, E., Shestakov, D.: Leveraging declarative languages in web application development. World Wide Web 19(4), 519–543 (2016)
66. Wakil, K., Jawawi, D.N.: Model driven web engineering: A systematic mapping study. e-Informatica Software Engineering Journal 9(1), 107–142 (2015)
67. Woo, M.: The rise of no/low code software development—no experience needed? Engineering 6(9), 960–961 (2020)
68. Yamouni Khelifi, N., Śmiałek, M., Mekki, R.: Generating database access code from domain models. In: 2015 Federated Conference on Computer Science and Information Systems. pp. 991–996 (2015)

Kamil Rybiński is Adjunct Professor at the Institute of the Theory of Electrical Engineering and Applied Informatics of the Warsaw University of Technology. He obtained his Ph.D. in software engineering from the Faculty of Electrical Engineering at the same university. His research interests includes requirements engineering, model-driven software development and knowledge representation.

Michał Śmiałek is Professor of Software Engineering at the Warsaw University of Technology. He obtained his habilitation (higher doctorate) degree in informatics from the Warsaw Military University and graduated from the Warsaw University of Technology (M.Sc. and Ph.D.) and the University of Sheffield (M.Sc.). Since 1991 he has worked in the industry as Software Developer, Project Manager and Professional Coach. His current research interests include model-driven software development, requirements engineering, software reuse, software language engineering and large-scale computing. He published and edited several books and over 100 papers in various journals and conference proceedings. He coordinated two European-level research projects, chaired several international conferences and reviewed for major computer science journals.

Received: July 15, 2023; Accepted: May 10, 2024.