A Cross-Disciplinary Language for Change Propagation Rules

Kiana Busch[1], Dominik Werle[1], Martin Löper[1], Robert Heinrich[1], Ralf Reussner[1], Birgit Vogel-Heuser[2]

Abstract— **Automated production systems are in operation for a long time and are continuously being changed. Therefore, for these systems it is important to have the ability to react efficiently to changes. Change propagation analysis approaches allow predicting the effects of changes before they are actually implemented. Such approaches often use predefined change propagation rules that indicate how the change propagates in a system. However, the change propagation rules used by these approaches are limited to a discipline such as information systems, to the structure of system elements in a discipline, or to a programming language such as Java. In this paper, we present a cross-disciplinary language to specify change propagation rules. The proposed language is independent of a particular discipline, structure of system elements, or programming languages. To show the improvement of the readability and the coverage of the change propagation rules with our language, we apply it to two existing approaches to change propagation analysis for the electronic and mechanical components, as well as control software of automated production systems.**

I. INTRODUCTION

One of the main characteristics of automated Production Systems (aPS) is their longevity. Longevity requires continuous changes to aPS in order to adapt to the changes in requirements or environment [1]. The measure of the ease to change a system is maintainability [2]. Thus, maintainability is one of the most important quality attributes of aPS. As aPS consist of electronic and mechanical components, as well as software systems, the elements from different disciplines influence each other during the evolution [1]. The mutual dependencies between the elements from different disciplines increase the complexity of the analysis of the aPS evolution.

To support evolution in aPS, change propagation approaches can be used to analyze the impact of a change at design time and before the change is actually implemented in the system. These approaches usually use a set of pre-defined *change propagation rules* to predict the change in aPS [1]. A change propagation rule specifies how the change propagates between two structural elements of a system in a specific discipline [3]. The possible types of structural elements and their relations are usually described by *modeling languages*, often also referred to as *metamodels*. A concrete system is represented by an *instantiation* of the modeling language (i.e., a *model*). Thus, a change propagation rule is most commonly described for all systems in the discipline of interest. Therefore, the change propagation rules depend highly on the metamodel. In order to be able to define the change propagation rules, the *domain expert* in aPS requires not only information about the system under study, but also information about the software architecture of the change propagation approach and more general programming skills. These factors make the change propagation analysis in aPS more complex and time-consuming. It is preferable to domain experts to focus only on the maintainability relationships between the elements of aPS, while specifying change propagation rules. Thus, it is desirable to have a language to specify change propagation rules that can be flexibly adapted to other systems, be reused, or adapted when the aPS metamodel changes.

There are already model-based approaches that use a set of pre-defined rules to predict the propagation of change within requirements or from the requirement to architectural elements at design time (e.g., [4], [5]). However, these approaches are limited to the propagation of change within the discipline of information systems. There are further approaches (e.g., [6], [7]) that are based on Unified Modeling Language (UML) and constraints. These approaches check the constraints, after the model changes. If all constraints are valid, the new model is in a consistent state. However, these approaches are only applicable to systems that can be modeled using certain UML diagram types. Further, the previously described approaches use a set of change propagation rules that cannot be flexibly adapted to other systems or metamodels, as they are developed for a specific metamodel. The approaches that define a flexible language for change propagation analysis instead of constraints (e.g., [8]) are also limited to UML class diagrams and, thus, not applicable to any aPS metamodels.

In this paper, we present a Domain-Specific Language (DSL) for specifying change propagation rules, which can be flexibly used in different disciplines. Thus, a DSL is particularly relevant for the aPS discipline, as aPS consist of further sub-disciplines, namely electronic and mechanical components, as well as software systems. Using a DSL for change propagation rules reduces code redundancy and improves readability and maintainability. Further, our DSL allows domain experts to describe the change propagation rules in a discipline without knowledge of specific constructs of a General Purpose Language (GPL) such as Java or the software architecture of a change propagation approach. Additionally, the change propagation rules specified in the DSL can be easily and flexibly adapted to new or changing aPS metamodels. The contributions of the paper are as follows: *i)* We present a DSL for specifying change propagation rules that can be applied regardless of the discipline under

[1]The Authors are with Department of Informatics, Karlsruhe Institute of Technology (KIT), 76131 Karlsruhe, Germany {kiana.busch, dominik.werle, heinrich, reussner}@kit.edu, martin.loeper@student.kit.edu

[2]The Author is with Department of Mechanical Engineering, Technical University of Munich (TUM), 85748 Garching, Germany {vogel-heuser}@tum.de

study, such as aPS or information systems. *ii)* We show the applicability of the DSL for two previously existing change propagation analysis approaches for the electronic and mechanical components, as well as for the control software for Programmable Logic Controllers (PLCs), which are programmed according to the International Electrotechnical Commission (IEC) 61131-3 standard [9]. Both approaches are realized in Java.

The remainder of this paper is organized as follows: Sec. II summarizes the state of the art. In Sec. III, we introduce our cross-disciplinary language for change propagation rules. We present the evaluation of our language in Sec. IV. Sec. V concludes the paper and presents future work.

II. State of the Art

Rule-based approaches to change propagation analysis are already used in many disciplines such as information systems: In [4], the authors present an approach to change propagation and consistency checking for requirements. Their approach uses rules to perform change propagation for different types of changes depending on the relation (e.g., inclusion) between two requirements. Göknil et al. [5] describe an approach to tracing changes between requirements and architectural elements. In context of embedded real-time systems, the authors of [10] propose a rule-based approach to the propagation of requirements change for Architecture Analysis and Design Language (AADL) models. The change propagation rules specify how the architecture is to be changed after a change to requirements. Dam et al. [6] present an approach to change propagation in software systems at a very high abstraction level based on rules in Object Constraint Language (OCL), which can be violated by a change. In contrast to our language, the OCL rules define constraints for a consistent state and can be checked on changes. This requires a much more specific description of dependencies in the models of interest as opposed to giving a conservative estimate of the change impact set. Another approach to identification of changes between two versions of a UML model is presented in [7]. The approach determines the affected model elements based on change propagation rules in OCL. To this end, predefined propagation rules for various UML diagrams and their associated elements are provided. In contrast to our approach, all of the introduced techniques implement rules in imperative code and are tailored to a specific modeling language, such as UML, without providing means to adapt the analysis to other modeling languages or disciplines. In [8], a rule-based approach to model-based change propagation analysis is presented. To do this, the authors define a DSL to describe change propagation rules for UML class diagrams. However, we provide a cross-disciplinary language for change propagation rules regardless of the type of change. Our DSL can be used to tailor rules to a specific metamodel for describing the systems in a discipline.

In addition to the discipline of information systems, there are rule-based approaches to change propagation in other disciplines such as business processes: Sunkle et al. [11] present an approach to change propagation in enterprise architecture based on relationships between the architectural elements. For this purpose, [12] introduces an approach that defines change propagation rules based on the semantics of relationships in a metamodel. However, the defined change propagation rules depend on the discipline and the underlying metamodel. Similar to approaches in information systems domain experts cannot implement change propagation rules without knowledge of the architecture of the approaches.

III. A Language for Change Propagation rules

First, we introduce a running example in the aPS context consisting of simplified change propagation relationships and how the relationships can be expressed in a GPL (i.e., Java). We then present our language design and the application of our language to the running example.

A. Running example

In order to illustrate our language, we use simplified examples of a metamodel of aPS [1].

Sensor example – forward reference: A sensor has physical interfaces, as shown in Fig. 1. By changing the sensor, its physical interfaces may be affected. Thus, the corresponding change propagation rule in Java searches along the *forward reference* in the metamodel. The code for this change propagation rule searches for the affected sensor in all instances of the metamodel to identify the connected physical interfaces, as presented in Listing 1.

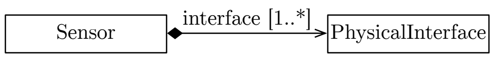

Fig. 1. Relationship between a sensor and its physical interface - An excerpt from an aPS metamodel

Crane example – backward reference: In this example, cranes are mounted on a physical table, as presented in Fig. 2. By replacing the table, it is necessary to demount the crane and mount it on the new table. The code of the change propagation rule needs to search along the *backward reference* to identify the affected crane. The code in Listing 2 searches for all cranes in all instances of the metamodel and filters them for a connection to the affected table.

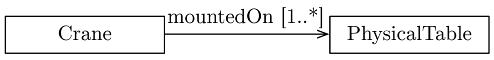

Fig. 2. Relationship between a crane and the physical table – An excerpt from an aPS metamodel

Bus system example – composition of forward and backward references: A change propagation rule can be more complex, if we have to search along several forward and backward references, as shown in an excerpt from a bus system in Fig. 3. By changing a bus slave, the change propagation rule in Listing 3 identifies all affected bus cables.

As the previous examples show, there are classes of change propagation rules involving forward and backward references that can be combined to complex change propagation rules. The code for expressing these complex rules either contains

```
public static List<PhysicalInterface> lookUpPhysicalInterfaceWithSensor(APS aPSModel,Collection<Sensor> modifiedSensor){
  List<PhysicalInterface> physicalInterfaces = new LinkedList<PhysicalInterface>();
  for (Sensor sensor : modifiedSensor)
    physicalInterfaces.addAll(sensor.getInterface());
  return physicalInterfaces; }
```

Listing 1. Java code for the lookup corresponding to the *Sensor Example*

```
public static List<Crane> lookUpCraneWithPhysicalTable(APS aPSModel, Collection<PhysicalTable> modifiedPhysicalTable) {
  List<Crane> cranes = new LinkedList<Crane>();
  for (Crane crane : aPSModel.getContainedCranes())
    for (PhysicalTable physicalTable : crane.getMountedOn())
      if (modifiedPhysicalTable.contains(physicalTable))
        cranes.add(crane);
  return cranes; }
```

Listing 2. Java code for the lookup corresponding to the *Crane Example*

```
public static List<BusCable> lookUpBusCableWithBusSlave(APS aPSModel, Collection<BusSlave> modifiedBusSlave) {
  List<BusCable> busCables = new LinkedList<BusCable>();
  for (BusCable busCable : aPSModel.getContainedBusCables())
    for (SignalPlug2 signalPlug : busCable.getPlug())
      for (SignalInterface signalInterface : signalPlug.getUses())
        for (SignalPlug1 modifiedSignalPlug : modifiedBusSlave.getPlug())
          for (SignalInterface modifiedSignalInterface : modifiedSignalPlug.getUses())
            if (modifiedSignalInterface.equals(signalInterface))
              busCables.add(busCable);
  return busCables; }
```

Listing 3. Java code for the lookup corresponding to the *Bus System Example*

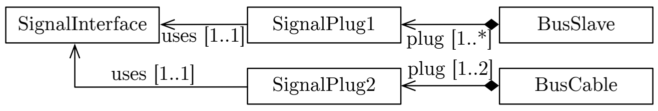

Fig. 3. Relationship between the elements of a bus system – An excerpt from an aPS metamodel

redundant code or requires additional effort for factoring out common code. Both options are error-prone.

B. Language Design

The main requirement for the Change Propagation Rule Language (CPRL) is the separation of concerns between the *domain expert*, who specifies the rules, and the *framework designer*, who implements change propagation tools. Additionally, we aim for a succinct and unambiguous syntax to keep the work needed for creating and maintaining the change propagation rules at a minimum.

The design of CPRL is motivated by scenarios built in previous work on change propagation analysis, specifically from information systems [13] and business processes [12] disciplines. While analyzing these approaches, we found that a large amount of change propagation rules can be expressed by specifying references between model elements and the direction that changes propagate along this reference. In the sensor example in Fig. 1, the reference *interface* that points from a *Sensor* to a *PhysicalInterface* is declared in a propagation rule. Further, change propagation in the opposite direction of a reference is also needed, as illustrated in Fig. 2.

CPRL is designed as a declarative language. The forward and backward change propagation along references illustrate a benefit of the *declarative* nature of the rule language (i.e., describing *what* has to be done) as opposed to implementing the change propagation in an *imperative* language (i.e., describing *how* to do it). To illustrate this benefit of declarative languages, let us consider the crane example. There are different implementations for finding model elements (i.e., *Crane*) that have a reference of a given type (i.e., *mountedOn*) to another given model element (i.e., *PhysicalTable*). For example, this could be implemented by traversing all model elements when a change in the *PhysicalTable* is detected. In a declarative language, the concrete implementation of the change propagation is hidden from the domain expert. The backward propagation along references also shows a further benefit of a DSL with an explicit parser and grammar: detecting all types of references for which opposite elements need to be indexed is more difficult for arbitrary GPL code whereas it can be trivially done during the parsing of the DSL.

C. DSL implementation

In this section, we will first shortly introduce the language-engineering framework that we chose for our approach. We also explain central elements of the language grammar.

Xtext[1] is a language engineering framework which allows the design and implementation of editing tools for textual DSLs based on a grammar of the language that is similar to the Extended Backus–Naur Form (EBNF). In particular,

[1]https://www.eclipse.org/Xtext/

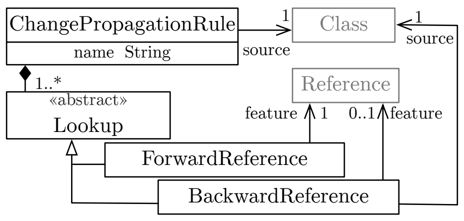

Fig. 4. Reduced illustration of the syntax of the CPRL.

plug-ins for the Eclipse[2] Integrated Development Environment (IDE) can be generated. We chose Xtext, as it is tightly integrated with the Eclipse Modeling Framework (EMF)[3] – a modeling framework for code generation.

Fig. 4 shows a reduced model of a CPRL rule. Class and Reference are not part of the language but types from EMF. Each rule has a unique *name*. Changes on a model element of the *source* type trigger the rule which results in the evaluation of an ordered list of *lookups* for determining the set of impacted model elements. Each lookup operates on the currently relevant set of elements, which we call the *current-set* in the following, and manipulates it in some way. The *current-set* is initialized as the source element that triggers the rule execution. In each step, the *current-set* is manipulated by looking up further elements that are related to the elements in the current set. Finally, the framework creates change impact markers for all elements in the set. The markers contain references to the source elements that triggered the change, so called *causing entities*.

The textual syntax for describing the rules is as follows:
`rule RULENAME: SOURCETYPE LOOKUPS`

`RULENAME` is the name the rule is referenced as. `SOURCETYPE` is the metaclass from which changes originate. `LOOKUPS` is a sequence of `FORWARDREFERENCE`s or `BACKWARDREFERENCE`s. `FORWARDREFERENCE`s are expressed as `-> FEATURE`. `BACKWARDREFERENCE`s are expressed as `<- SOURCE[FEATURE]`. If the feature is omitted, all references from the `SOURCE` type are analyzed.

The design of the language and the editor support provided by Xtext allow for different kinds of validation for the language, particularly type safety. For example, it is only possible to declare references that originate from the type of the *current-set* when specifying a forward propagation rule. In the sensor example, it would not be possible to declare a reference other that *interface* for a source type of *Sensor*. After this forward propagation is declared (i.e., the type of the *current-set* is subsequently *PhysicalInterface*), only attributes and references of this type can be used for further lookup methods. These type checks are supported by the editing tools by only suggesting appropriate references and types to the user.

The language supports further features, which we do not further describe in more detail here: elimination of duplicates in the *current-set*, changing the *causing entities*, filtering by a specific type, and referencing other change propagation rules.

[2]`https://www.eclipse.org/eclipse/`
[3]`https://www.eclipse.org/modeling/emf/`

D. Application to the Running Example

Listing 4 illustrates the application of our approach to the running example defined in subsection III-A. The rules defined in lines 1, 2, and 3 correspond to the Sensor, Crane, and Bus System examples, respectively. In comparison to the code, the given rules are less verbose and act in the domain of the modeling language, e.g., by referring to the reference by the name `interface` (Listing 4, Line 1) instead of using a call to the generated method `getInterface()` (Listing 1, Line 4). Furthermore, there is no need to make the cardinality of references explicit in the rule language, as it is the case with GPL code. For example, there is no syntactic difference between the reference from `Crane` to `PhyiscalTable` in CPRL (i.e., `mountedOn` reference in Listing 4, Line 1) and the reference from `SignalPlug2` to `SignalInterface` (i.e., `uses` reference in Listing 4, Line 3). In GPL code, however, the domain expert needs to differentiate between a method call that returns a collection that needs further iteration (e.g., `getMountedOn()` in Listing 2, Line 4) and a call that returns exactly one element (e.g., `getUses()` in Listing 3, Line 5). Altogether, the description is lifted from the generated GPL code to the modeling language level.

IV. Evaluation

This section shows the applicability of our DSL. We apply our DSL to two approaches to change propagation analysis.

A. Study Design

To evaluate CPRL, we follow a Goal Question Metric (GQM) plan [14]. **Goal 1** is to evaluate the coverage of CPRL when specifying the change propagation rules in a specific discipline. It is important to note, that the goal of the evaluation is not to evaluate the change propagation approaches. Each approach [1], [15] has been already evaluated. To evaluate the coverage of the language, we define **Question 1**: How well can CPRL cover the change propagation rules in a change propagation approach? In order to answer this question, we define **Metric 1** as the ratio of the number of change propagation rules that can be expressed with CPRL to the number of total change propagation rules that exist in an existing change propagation approach.

We formulate a further goal, which addresses the well-known benefits of DSL regarding less redundant code and less technical code compared to a GPL. **Goal 2** analyzes the readability of the change propagation rules specified in Java compared to CPRL. To evaluate this goal, we use **Question 2**: How can CPRL help domain experts understand existing change propagation rules and formulate new ones. To answer this question in a measurable way, we calculate **Metric 2** as the average lines of code for the existing change propagation rules in Java and compare it to CPRL.

To evaluate CPRL we chose the aPS discipline. This discipline contains heterogeneous elements such as electronic and mechanical components, as well as software. Thus, it covers several sub-disciplines [1]. We apply CPRL to two existing rule-based approaches: *i)* an approach to analysis of the change propagation in mechanical and electronic

```
rule SensorToInterface:      aps::Sensor  > interface;
rule PhysicalTableWithCrane: aps::PhysicalTable <  aps::Crane[mountedOn];
rule BusCableFromBusSlave:   aps::BusSlave  > plug  > uses <  aps::SignalPlug2[uses] <  aps::BusCable[plug];
```

Listing 4. CPRL rules for the running example.

TABLE I
OVERVIEW OF RULES FOR MECHANICAL AND ELECTRONIC PARTS: CHANGE PROPAGATION FROM METACLASS A TO METACLASS B

Metaclass A	**Metaclass B**
Component	Structure, Module, Interface
Module	Structure, Module, Component, Interface
Structure / Interface	Module, Component

components of aPS [1] and *ii)* an approach to analysis of the change propagation in PLC software [15] that is programmed according to the IEC 61131-3 standard [9]. The rules of both approaches were implemented in Java. We used Java, as a program in Java requires almost as many or fewer lines of code than programs in other programming languages [16]. In the evaluation we specify existing change propagation rules of both approaches in CPRL.

B. Change Propagation Rules

This section gives an overview of the change propagation rules already implemented by both approaches.

1) Electronic and Mechanical Components of aPS: The change propagation approach for electronic and mechanical components is based on the metamodel describing the structure of a plant in aPS. The metamodel can be constructed at different abstraction levels. On a higher abstraction level, the metamodel is comprised of components, modules, structures, and interfaces. A component represents parts that can be supplied by a third party, while a module can be assembled by a plant manufacturer. Components and modules can have interfaces for fixation and communication. Structures can be used to organize modules, components, and interfaces [1].

Each row in Table I defines a change propagation rule for the metamodel elements (i.e., metaclasses). Each rule describes the propagation of change from metaclass A (i.e., the metaclass in the first column of a row) to metaclass B (i.e., metaclasses in the second column of the same row). An example of a rule is the change propagation from components to interfaces. In addition, the approach iterates over all change propagation rules until no new elements in the current iteration are marked as changed (i.e., transitive closure).

2) Control Programs According to the IEC Standard: The change propagation approach for PLC software [15] is based on a metamodel for control programs according to IEC standard. The metamodel contains elements that are relevant for analyzing maintainability. A program is comprised of other elements such as global variables, functions, and function blocks. Global variables can be used for input and output. Function Blocks are stateful, as they can access global variables, while functions perform stateless calculations [9].

The rows of Table II contain the change propagation rules. The rules describe the change propagation from metaclass A to metaclass B. While the first column of Table II shows the candidates for metaclass A, the second column shows the candidates for metaclass B. An example of a rule is the change propagation from a global variable to the corresponding function block. This approach also calculates a transitive closure of the affected elements. To this end, if a function block or an interface causes a change propagation, the analysis calculates the change propagation using the change propagation rules. Then, it iterates over the newly inserted changing elements and calculates the changes based on the newly inserted changing elements and the change propagation rules. In all other cases, the analysis aborts the change propagation after one iteration.

TABLE II
OVERVIEW OF RULES FOR IEC SOFTWARE: CHANGE PROPAGATION FROM METACLASS A TO METACLASS B

Metaclass A	**Metaclass B**
Global Variable	Configuration, Function Block, Method, Program
Function	Function, Function Block, Method, Program
Function Block	Abstract Method, Abstract Property, Function, Function Block, Global Variable, Method, Program, Property
Interface	Abstract Method, Abstract Property, Function, Function Block, Global Variable, Interface, Method, Program, Property
Abstract Property	Function Block, Method, Program, Property
Abstract Method / Method / Property	Function Block, Method, Program

C. Results

To answer **Question 1**, we use CPRL to specify the existing change propagation rules of both approaches, which have been already realized in Java. The existing change propagation rules are either a change propagation along a forward reference or a backward reference. Thus, these change propagation rules can be covered by CPRL. However, a change propagation rule in Java does not always correspond to a change propagation rule in CPRL. For example, some change propagation rules in Java had to be split into more than one change propagation rule in CPRL. In contrast, there are also change propagation rules in Java that were combined into one change propagation rule in CPRL.

To construct the transitive closure, the approaches iterate over all model elements that are marked by change propagation rules as changed and apply the change propagation rules. CPRL does not currently support loops over all change propagation rules. Thus, the coverage of CPRL for approach for mechanical and electronic components and for software (i.e., **Metric 1**) is approximately 93% and 98%, respectively.

To answer **Question 2**, we count the lines of code that are in average needed to specify a change propagation rule in Java and in CPRL. While counting the lines of code in Java, we ignored blank lines, comments, and lines containing only braces. For example, the code of the bus system example (cf. Listing III-A) has 10 lines. Further, the help methods in Java for the change propagation rules were counted only once. Then, we specified all existing rules in both change propagation approaches with CPRL. We consider only change propagation rules that can be specified by both Java and CPRL. A Java rule for approaches to change propagation analysis for mechanical and electronic components and for IEC programs (i.e., **Metric 2**) has 12 and 10 lines of code in average, respectively. Using CPRL we could specify the change propagation rules in 1 line in average. The reason for longer rules in Java is the technical constructs in a GPL such as loops or branches leading to a redundant code in Java.

D. Limitations

CPRL currently supports forward and backward references, as well as their combination. As described previously, these types and their combinations allow for the coverage for a majority of change propagation rules in the disciplines under study. More complex language features can be possible extensions for further types of lookup methods. Currently, these types of lookup methods can be integrated using imperative code written in Java. Although we currently do not provide these features, this is not a conceptual restriction of CPRL and will be implemented in future work.

E. Threats to Validity

We consider the following categories of validity [17]:

Internal validity: We applied CPRL to the change propagation rules of two impact analysis approaches. The results of the evaluation depend on the implementation of the change propagation rules. Thus, both impact analysis approaches and CPRL were carried out by different developers.

External validity: We evaluated CPRL by application to change propagation rules in aPS. Thus, the results might not be generalizable to other disciplines. But, the discipline of aPS is composed of other sub-disciplines such as electronic, mechanical, and software. Thus, we showed the applicability of our DSL to several disciplines. The focus is to show the relevance of a DSL to reduce the amount of technical code and to improve the readability of change propagation rules. Further, we applied our DSL to existing approaches, that are implemented independently of the DSL. Both approaches are evaluated in the respective sub-discipline.

Construct validity: In approaches under study in evaluation, the CPRL does not cover iterations over all change propagation rules. However, we did not select the approaches with focus on this criterion. The application of CPRL to these approaches demonstrates that using a forward and a backup reference, as well as their combinations allows for the coverage for a majority of change propagation rules.

Conclusion validity: To minimize the interpretation effects of individual researchers we used metrics for the evaluation. The lines of code metric can be different in different programming language. We used Java, as a program in Java requires almost as many or fewer lines of code than programs in other programming languages [16]. We counted the lines that contribute to change propagation rules (e.g., we ignored comments or lines of code containing only a brace).

V. Conclusion & Future Work

In this paper, we introduced CPRL, a language for the declaration of change propagation rules. The presented language facilitates the separation of concerns in the engineering process for change propagation approaches. It allows domain experts to specify rules without making the way they are evaluated or interpreted explicit. The design process of CPRL is driven by a previously evaluated set of rules for the change propagation in the disciplines of information systems and business processes. The evaluation of CPRL demonstrates that the DSL can be used to specify most change propagation rules independently of the discipline under study by applying it to the discipline of aPS.

Future work includes the application of CPRL to further disciplines and augmenting CPRL with additional features to include more complex change propagation rules such as calculating transitive closure.

Acknowledgment

This work was partially supported by the DFG under the Priority Programme SPP1593 (RE1674/12-1, VO937/29-1) and as part of the Research Training Group GRK 2153: Energy Status Data – Informatics Methods for its Collection, Analysis and Exploitation.

References

[1] B. Vogel-Heuser *et al.*, "Maintenance effort estimation with kamp4aps for cross-disciplinary automated production systems - a collaborative approach," in *20th IFAC World Congress*, Toulouse, France, 2017.

[2] ISO/IEC, "ISO/IEC 25010 - Systems and software engineering - Systems and software Quality Requirements and Evaluation (SQuaRE) - System and software quality models," Tech. Rep., 2010.

[3] R. Heinrich *et al.*, "A methodology for domain-spanning change impact analysis," in *Euromicro Conference on SEAA*. IEEE, 2018.

[4] A. Göknil *et al.*, "Change impact analysis for requirements: A metamodeling approach," *Information and software technology'14*, vol. 56.

[5] A. Göknil *et al.*, "A rule-based change impact analysis approach in software architecture for requirements changes," *CoRR*, vol. 1608.02757, 16.

[6] H. K. Dam *et al.*, "An agent-oriented approach to change propagation in software maintenance," *JAAMAS*, vol. 23, no. 3, pp. 384–452, 2011.

[7] L. C. Briand *et al.*, "Impact analysis and change management of uml models," in *ICSM*, 2003.

[8] K. Müller and B. Rumpe, "A model-based approach to impact analysis using model differencing," *CoRR*, vol. abs/1406.6834, 2014.

[9] IEC, "61131-3: Programmable controllers–part 3: Programming languages."

[10] A. Göknil *et al.*, "A rule-based approach for evolution of AADL models based on changes in functional requirements," in *ECSA Workshops*. ACM, 2016, p. 10.

[11] S. Sunkle *et al.*, *Analyzing Enterprise Models Using Enterprise Architecture-Based Ontology*. Springer, 2013, pp. 622–638.

[12] K. Rostami *et al.*, "Architecture-based Change Impact Analysis in Information Systems and Business Processes," in *ICSA*. IEEE, 2017.

[13] K. Rostami *et al.*, "Architecture-based assessment and planning of change requests," in *QoSA*. ACM, 2015, pp. 21–30.

[14] V. R. Basili *et al.*, "The goal question metric approach," in *Encyclopedia of Software Engineering*. Wiley, 1994.

[15] J. Rätz, "Erweiterung eines Wartbarkeits-Frameworks für die Programmiersprache IEC 61131-3," 2017.

[16] L. Prechelt, "An empirical comparison of seven programming languages," *Computer*, vol. 33, no. 10, pp. 23–29, 2000.

[17] P. Runeson *et al.*, *Case Study Research in Software Engineering: Guidelines and Examples*, 1st ed. Wiley, 2012.