

# Understanding the role of Model Transformation Compositions in Low-Code Development Platforms

Apurvanand Sahay  
apurvanand.sahay@univaq.it  
Università degli Studi dell'Aquila  
L'Aquila, Italy

Davide Di Ruscio  
davide.diruscio@univaq.it  
Università degli Studi dell'Aquila  
L'Aquila, Italy

Alfonso Pierantonio  
alfonso.pierantonio@univaq.it  
Università degli Studi dell'Aquila  
L'Aquila, Italy

## ABSTRACT

Low-code development platforms (LCDPs) permit developers that do not have strong programming experience to produce complex software systems. Visual environments permit to specify workflows consisting of sequential or parallel executions of services that are directly available in the considered LCDP or are provided by external entities. Specifying workflows involving different LCDPs and services can be a difficult task. In this paper, we propose the adoption of concepts and tools related to the composition of model transformations to support the specification of complex workflows in LCDPs. We elaborate on how LCDPs services can be considered as model transformations and thus, workflows of services can be considered as model transformation compositions. The architecture of the environment supporting the proposed solution is presented.

## CCS CONCEPTS

• **Software and its engineering** → **Abstraction, modeling and modularity**; **Model-driven software engineering**.

## KEYWORDS

Model Driven Engineering, Low-Code Development Platform, Model Transformation, Model Transformation Composition

### ACM Reference Format:

Apurvanand Sahay, Davide Di Ruscio, and Alfonso Pierantonio. 2020. Understanding the role of Model Transformation Compositions in Low-Code Development Platforms. In *ACM/IEEE 23rd International Conference on Model Driven Engineering Languages and Systems (MODELS '20 Companion)*, October 18–23, 2020, Virtual Event, Canada. ACM, Montreal, Canada, 5 pages. <https://doi.org/10.1145/3417990.3420197>

## 1 INTRODUCTION

Low-Code Development Platforms (LCDPs)<sup>1</sup> are visual environments that are being increasingly promoted by major IT players for supporting *citizen developers* to create software systems even if they lack programming background and knowledge [16]. One of the most prominent application domain for LCDPs is *process automation* [10]: citizen developers are provided with visual environments to specify workflows orchestrating sequential or even

<sup>1</sup>Hereafter, the terms *low-code platforms* and *low-code development platforms* are used interchangeably and are abbreviated as *LCDPs*.

*MODELS '20 Companion*, October 18–23, 2020, Virtual Event, Canada

© 2020 Association for Computing Machinery.

This is the author's version of the work. It is posted here for your personal use. Not for redistribution. The definitive Version of Record was published in *ACM/IEEE 23rd International Conference on Model Driven Engineering Languages and Systems (MODELS '20 Companion)*, October 18–23, 2020, Virtual Event, Canada, <https://doi.org/10.1145/3417990.3420197>.

parallel consumptions of services, each typically provided by external providers, which the used LCDP is able to connect. Thus, developers can specify processes e.g., to retrieve data from external data sources (e.g., calendar, sensors, and files stored in cloud services), to manipulate retrieved data. It can be achieved by means of the provided facilities or even by using external services, and to perform some aggregation and analysis according to rules defined with the languages provided by the platform. However, when complex workflows have to be specified, developers have to be aware of the possible service providers that the used LCDP is able to interact with, and the manipulation means that might be exploited to finally develop the desired process.

In this paper, we focus on the low-code development of complex workflows involving the interoperability of different services. In such a setting, we aim at providing citizen developers with the means to declaratively specify the goals of the desired processes and to get back the services, and the corresponding interactions. Such interactions might be potentially employed to develop the process previously specified in a declarative manner at a higher level of abstraction. To this end, we investigate the possibility of relying on the results developed by the Model Driven Engineering (MDE) [11] research community about the topic of *model transformation composition* [2]. The envisioned idea is considering workflows as corresponding model transformation chains, and the services to be consumed in the workflows as model transformations that have to be properly composed. Thus, many smaller and simpler model transformations are composed together to realize complex workflows to be executed in the considered LCDP. The contribution of the paper is to propose a research direction to support several specifications of services to define and execute their complex workflows in LCDPs.

The paper is structured as follows: Section 2 introduces the problem we want to cope in this paper. Section 3 presents an overview of existing approaches for composing model transformations. Section 4 proposes their adoption to support the definition of complex workflows in LCDPs. Section 5 concludes the paper and presents our future plans.

## 2 PROBLEM STATEMENT

LCDPs are visual environments that permit mainly in a declarative manner to develop complex applications even for developers that do not have strong programming experience. Systems like IFTTT<sup>2</sup>, and zapier<sup>3</sup> provide users with easy to use tools for developing workflows consisting of sequential or even parallel execution of services. By means of such tools, users can specify workflows that

<sup>2</sup><https://ifttt.com/>

<sup>3</sup><https://zapier.com/>

can be triggered e.g., when a new email arrives and a corresponding spreadsheet is updated in order to keep track of emails that have been sent from specific addresses. Even in Internet of Things (IoT) scenarios, LCDPs can play a key role e.g., when devices from different brands and ecosystems need to be integrated in a same physical environment.

Workflows that users might have the need to specify can be complex and involve different platforms and services. An explanatory scenario is shown in Figure 1 that considers two LCDPs named as *LCDP1* and *LCDP2*. In *LCDP1*, there are two applications shown as circles called *App11* and *App12*. Inside *App11*, there are two services shown as squares named as *x1* and *x2*. Inside *App12*, there are also two services shown in squares named *m1* and *m2*. Likewise, in *LCDP2*, there are two applications shown as circles called *App21* and *App22*. Inside *App21*, there are three services shown in squares called as *y1*, *y2* and *y3*. Similarly in *App22*, there are also three services shown in squares called as *z1*, *z2* and *z3*.

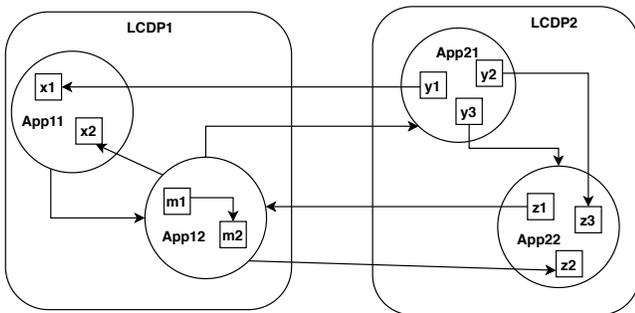


Figure 1: Service and Data Flows in LCDPs

According to the depicted arrows in Figure 1, different kinds of interactions (i.e., service invocations and application interfaces) can occur in workflows like the one previously described in the example involving Mendix and Appian. In the following points, we distinguish two kinds of interactions, i.e., *Inter LCDPs* and *Intra LCDPs*:

- *Inter LCDP*: This kind of interactions occurs when data from services and applications produced in one LCDP are manipulated or consumed by services and applications provided by another LCDP. Such manipulations or consumption are shown in arrows *y1* to *x1*, *z1* to *App12*, *App12* to *App21* and *App12* to *z2*.
- *Intra LCDP*: This kind of interactions occurs when data from services or applications are manipulated or consumed by services or applications within the same low-code platform. Such interactions can be further detailed as:
  - *Inter applications*: Data from one application are manipulated or consumed by different applications within the same LCDP. Such manipulations are shown in arrows *y2* to *z3*, *y3* to *App22*, *App11* to *App12* and *App12* to *x2*.
  - *Intra application*: Data from services in one application within the same LCDP are manipulated by another services offered by the same application. For instance, data produced by pre-built forms, reports, etc., are used and manipulated by the same application to provide different

view such as grid view, Kanban view, etc. Such manipulation is shown in *m1* to *m2*.

To make a concrete example with respect to Figure 1, let us suppose to work with the Mendix and Appian low-code development platforms as shown in Figure 2. In Mendix, there are two developed applications called *Attendance Management System* and *Autonomous Vacation Management System*. *Attendance Management System* determines the attendance of each employee on a given month and it provides a service named as *Enrollment System*, which is reused by the application known as *Enrollment Management System* built in the Appian low-code platform. Also, *Autonomous Vacation Management System* built in Mendix uses an application known as *Vacation Management System* built in Appian. *Autonomous Vacation Management System* interacts with an external service provided by IFTTT<sup>4</sup> to build an automatic allocation of the vacation if the employee asked for it. This example deals with different services and applications that interact across different LCDPs.

The data from *Autonomous Vacation Management System* application is used in *Attendance Management System* in Mendix to automate the absence of an employee due to the allotment of vacation. Also, suppose there is a service named as *Salary Section* in the application *Vacation Management System* which is determined by another service of another application *Enrollment Management System* known as *Grading System* within the same low-code platform Appian. Such interactions within the same LCDPs can also be executed within applications or across different applications of that LCDP.

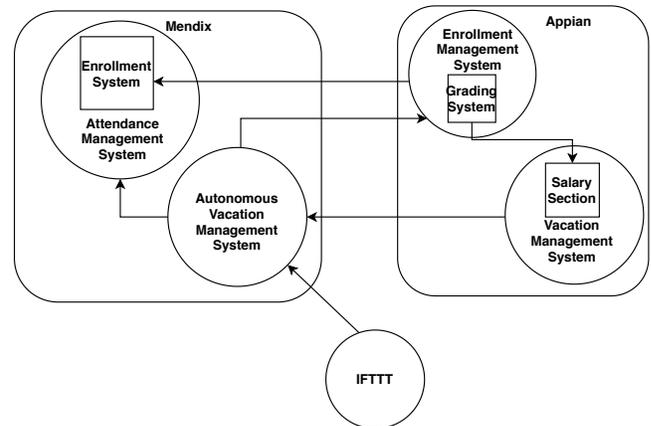


Figure 2: Example of Service and Data Flows in LCDPs

Specifying workflows involving different LCDPs and services can be a difficult task. To support their specifications, we investigate the possibility of relying on techniques and tools developed in the MDE field for chaining model transformations. In particular, if services are seen as model transformations, specifying workflows consisting of orchestrations of different services can be seen as the problem of properly chaining different model transformations. To this end, next section makes an overview of existing approaches for model transformation compositions. Afterwards, we propose to

<sup>4</sup><https://ifttt.com/>

employ them to support the development of complex workflows in LCDPs.

### 3 MODEL TRANSFORMATION COMPOSITION APPROACHES

Composing model transformations is a way to develop complex model management operations by composing smaller ones. There are two main ways to compose model transformations i.e., *internal* and *external*. In internal composition, two model transformation definitions give place to a new transformation. In external composition, two different transformations are chained together and the output models of the first transformation are given as input to the second one. Over the last years, several approaches have been proposed to compose model transformations and relevant works in such a context are overviewed in Section 3.1. The problem statement discussed in Section 2 indicates a clear need to research and develop tools to implement those workflows in concern with LCDPs. An explanatory example showing the usage of model transformation compositions is given in Section 3.2

#### 3.1 Overview

Basciani et al. (2018a, 2018b) [2, 3] propose an approach for the external composition of model transformations. In particular, given as input the source model, and the wanted target metamodel, the system is able to find out a ranked list of transformation chains with respect to a notion of information loss implemented in a customized Dijkstra algorithm.

Etien et al. (2015) [7] propose an approach to transform very large models by decomposing them based on a separation of concern technique and then use localized transformations to check the desired outcomes according to the objectives of the application. Thus the proposed technique consists of building localized transformations and combine them with the help of a composition language.

Aranega et al. (2012) [1] make use of feature models to automate a consistent set of model transformations and generate an executable chain implementing the desired objectives.

In [5], authors propose an approach to determine which chaining of available model transformations gives the desired result with respect to pre-conditions, post-conditions, and behavioural aspects of individual rules. Thus, commutativity of the chaining of model transformations is also used to detect identical results by using both sides of the transformation. Similarly, in [4] authors present an approach to find out the best transformation chain (that satisfies given chaining constraints) by statically analyzing single transformations.

In [6] authors propose a technique to combine independent model transformations that do not handle compatible source and target metamodels, and that might jointly work to achieve the wanted objective. The approach relies on the composition language for independent model transformations with incompatible meta-models.

Wagelaar et al. (2010, 2008) [14, 15] propose an internal composition technique named model superimposition that allows for extending and overriding rules in different transformation modules.

Vanfooff et al. (2006) [13] present a language to specify transformation chains. The language is based on UML activity diagrams

and it is independent from the languages used to develop the transformations being chained. Similarly, Rivera et al. (2009) [9] propose a graphical language for orchestrating ATL transformations. The language provides users with modeling constructs to specify conditional, parallel, and looping execution of different ATL model transformations.

#### 3.2 Explanatory example

We have mentioned external (chaining) as well as internal composition of model transformations for transforming and composing different services. According to paper [12], it is easier to build and test the smaller model transformations separately and then merge them together to form the whole complex transformation.

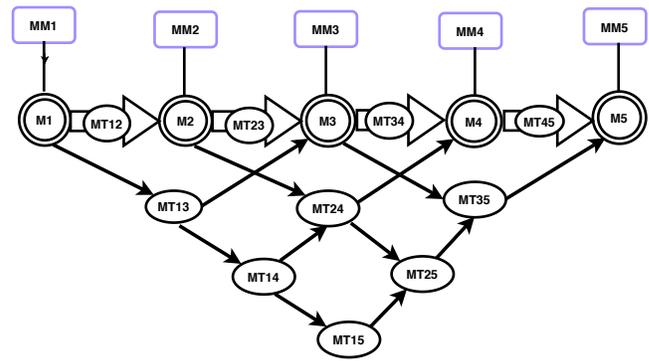


Figure 3: Graph of Model Transformation Compositions

Figure 3 graphically represents an example of possible compositions when different model transformations are available. Suppose, a grid view model is expected to be transformed into a Kanban view model which in turn is expected to be converted to a pie chart model. These models have their respective metamodels. Either a chaining or an internal combination of model transformations is expected to transform from a grid view model to a pie chart model. Actually, the grid view model may or may not be converted to intermediate Kanban view model based on the chaining or internal combination of model transformations, respectively.

The following points are derived from the graph of model transformation compositions shown in Figure 3. The metamodel X is shown as MMX indicated inside a rectangle shape. The model X of the corresponding metamodel is shown as MX indicated inside an enclosed circles shape. The model transformation which transforms from MMX to MMY is shown as MTXY indicated inside an oval shape. The arrows represent the forward usage of a model or model transformation to another model or model transformation. Model transformation can be formulized as:  $MTXY = MTX(Y-1) ++ MT(X+1)Y$ ; where '++' means composition (external or internal).

Figure 3 shows five models corresponding to their respective metamodels. The model transformation MT transforms one model to another. To transform model M1 to M5, the maximum number of required forward chainings of model transformations are 4 which is from MT12 to MT23 to MT34 to MT45; while the minimum number of MT required is only one which is MT15. This is done by the composition of several intermediate model transformations. Therefore,

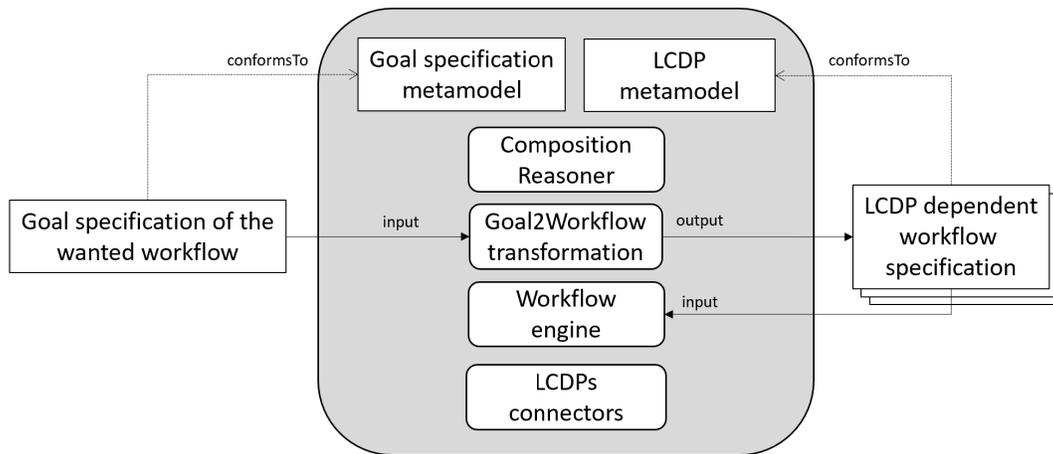


Figure 4: Proposed approach

maximum number of forward chainings of model transformations possible from source to target models are  $(n-1)$ ; where  $n$  is the number of the metamodels to be taken in transformations. The minimum number of model transformation composition, which is possible from source to target model is 1. This shows variety of paths of model transformations that are involved in converting one model to the another.

Also, from Figure 3, the total number of model transformations can be calculated as follows. If 2 models are available to be transformed, then only 1 model transformation is possible. Similarly, for 3, 4 and 5 models to be transformed, the total number of model transformations possible are 3, 6 and 10 respectively. Therefore, we can generalize this trend. The trend shows for  $n$  models (where  $n > 2$ ) to be transformed, there will be  $n(n-1)/2$  model transformations possible. Therefore, the total number of model transformations possible within a system of  $n$  models are  $n(n-1)/2$ . This explains the possible number of model transformations in specifying complex workflows available in a software and gives us the ability to find out the optimal composition of model transformations by estimating the optimal paths [3].

The paper aims at composing different model transformations (either, internally or externally) to have an optimal path to achieve the target workflows. Therefore, a transformation path is aimed with minimal cost and time. This would require all the possible number of model transformations available to obtain a specific workflow with different combinations of model transformation compositions. Such model transformation compositions are more reusable if the overlapping of their combination is maximum. This gives an efficient and simpler solution to define all the workflows because most of the compositions are pre-built and therefore it can be easily reused for a new workflow with minimum changes on the newer transformations. This maximum overlapping of model transformation compositions utilizes the use of pre-built compositions developed from previous workflows to deliver a new workflow.

## 4 PROPOSED APPROACH

In this section, we elaborate on how model transformation composition techniques can be used to support the specification of complex workflows in LCDPs. In particular, we see the possible interactions occurring *intra* and *inter* LCDPs presented in Section 2 as proper orchestrations of different services. Then, if we can manage such services as model transformations, then we can reuse the theories underpinning existing composition approaches for model transformations. In this respect, Figure 4 depicts the main components of the envisioned approach, which is detailed in the following.

*Goal specification metamodel:* We plan to employ a similar technique as proposed in [2] to chain model transformations. In that case, the goal specified by the user consists of the target metamodel that the desired model should conform. By considering the given goal and the input model, the approach is able to identify possible transformation chains, if any. In a similar manner, we aim at providing users of LCDPs with the means to specify the characteristics of the desired workflows at a high-level of abstraction. The tools and languages shall permit to specify constraints, functional, and non-functional requirements that the final workflow should satisfy. An example of goal is that the user wants to take some input model and visualize it by means of two target views, i.e., grid view and Kanban view. To enable the adoption of the metamodel, it is necessary to define a modeling language providing users with all the modeling constructs formalized in the metamodel.

*LCDP metamodels:* We plan to define metamodels for specifying characteristics of the supported LCDPs. The idea is to be able to specify workflow models that can be executed by the corresponding LCDP. The specification of such metamodels require the analysis of different platforms with the aim of identifying the peculiar characteristics with respect to the provided mechanisms for specifying and executing workflows [10]. The metamodels can be mainly classified as follows. The main data metamodel of the whole LCDP is mapped to view-specific metamodels. They are essentially a design-time view and a run-time view, which correspond to the static analysis of the data model before the deployment, and to run-time analysis of the data model after its deployment, respectively. Such views store

only those data that are relevant to its specific view [8]. The citizen developers should only see these view models individually and not the whole of LCDP's data model. These separations of views allow the citizen developer to focus on either of the views without much worrying about the overall expressiveness and flexibility of the data model of a particular LCDP.

*LCDPs connectors:* They are the software components that permit the system to connect to the different low-code development platforms by relying on provided APIs.

*Composition Reasoner:* Such a component checks the feasibility of the input goal with respect to the available services provided by the LCDPs which the system is able to connect. The list of such services is retrieved and kept updated by relying on the available LCDPs connectors. For instance, by considering simple goal specification previously given, the composition reasoner would check if the available LCDPs can manage services' view types like grid and Kanban views.

*Goal2Workflow transformation:* By relying on the outcome of the Composition Reasoner and by considering the available LCDPs connectors, this component generate possible workflows that are compatible with the specified goals. Similarly to what happens in model transformation compositions, different solutions can be possible and the system will provide the user with a ranked list. By considering the previous example, the component would show all the possible workflows that permit to generate grid and Kanban views out of a source model. In case there are more than one services (even provided by different LCDPs) are able to manage grid and Kanban view, the component would produce all the possible compositions.

*Workflow engine:* It is the software component able to execute models generated by the *Goal2Workflow transformation*. The engine interacts with the LCDPs that enable the execution of workflows via some exposed APIs.

## 5 CONCLUSION AND FUTURE WORKS

The paper gives preliminary study on the usage of model transformation compositions in the domain of low-code development platforms. The problem that we want to address in the medium term is about the specification of complex workflows in LCDPs. The goal is to support citizen developers by providing them with modeling constructs that permit to specify the goal of the desired workflows at a high-level of abstraction. By relying on the techniques and tools developed for composing model transformations, the idea is to generate possible workflows that satisfy the initial goal. The architecture of the planned solution has been overviewed and we plan to work on it by first focusing on the needed modeling languages, i.e., the goal and workflow specification languages. They have to be defined in an iterative manner by specifying real situations and refine the available constructs in case of errors or

to cover unforeseen requirements. As a next step, we will focus on the development of the workflow engine and all the dependent components including the reasoner, and the goal to workflow transformation. We plan to apply the proposed approach in different application domains including the development of IoT systems and the specification of complex workflows involving different model managements operations.

## ACKNOWLEDGEMENT

This work is funded by the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie – ITN grant agreement No 813884.

## REFERENCES

- [1] Vincent Aranega, Anne Etien, and Sebastien Mosser. 2012. Using feature model to build model transformation chains. In *International Conference on Model Driven Engineering Languages and Systems*. Springer, 562–578.
- [2] Francesco Basciani, Mattia D'Emidio, Davide Di Ruscio, Daniele Frigioni, Ludovico Iovino, and Alfonso Pierantonio. 2018. Automated selection of optimal model transformation chains via shortest-path algorithms. *IEEE Transactions on Software Engineering* (2018).
- [3] Francesco Basciani, Davide Di Ruscio, Mattia D'Emidio, Daniele Frigioni, Alfonso Pierantonio, and Ludovico Iovino. 2018. A tool for automatically selecting optimal model transformation chains. In *Proceedings of the 21st ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*. 2–6.
- [4] Raphaël Chenouard and Frédéric Jouault. 2009. Automatically discovering hidden transformation chaining constraints. In *International Conference on Model Driven Engineering Languages and Systems*. Springer, 92–106.
- [5] Anne Etien, Vincent Aranega, Xavier Blanc, and Richard F Paige. 2012. Chaining model transformations. In *Proceedings of the First Workshop on the Analysis of Model Transformations*. 9–14.
- [6] Anne Etien, Alexis Muller, Thomas Legrand, and Xavier Blanc. 2010. Combining independent model transformations. In *Proceedings of the 2010 ACM Symposium on Applied Computing*. 2237–2243.
- [7] Anne Etien, Alexis Muller, Thomas Legrand, and Richard F Paige. 2015. Localized model transformations for building large-scale transformations. *Software & Systems Modeling* 14, 3 (2015), 1189–1213.
- [8] Nick Jansen. 2019. *Exploring interactive application landscape visualizations based on low-code automation*. Master's thesis.
- [9] José E Rivera, Daniel Ruiz-Gonzalez, Fernando Lopez-Romero, José Bautista, and Antonio Vallecillo. 2009. Orchestrating ATL model transformations. *Proc. of MtATL 9* (2009), 34–46.
- [10] Apurvvanand Sahay, Arsene Indamutsa, Davide Di Ruscio, and Alfonso Pierantonio. 2020. Supporting the understanding and comparison of low-code development platforms. In *2020 46th Euromicro Conference on Software Engineering and Advanced Applications*.
- [11] Douglas C Schmidt. 2006. Model-driven engineering. *Computer-IEEE Computer Society-* 39, 2 (2006), 25.
- [12] Shane Sendall and Wojtek Kozaczynski. 2003. Model transformation: The heart and soul of model-driven software development. *IEEE software* 20, 5 (2003), 42–45.
- [13] Bert Vanhooff, Stefan Van Baelen, Aram Hovsepian, Wouter Joosen, and Yolande Berbers. 2006. Towards a transformation chain modeling language. In *International Workshop on Embedded Computer Systems*. Springer, 39–48.
- [14] Dennis Wagelaar. 2008. Composition techniques for rule-based model transformation languages. In *International Conference on Theory and Practice of Model Transformations*. Springer, 152–167.
- [15] Dennis Wagelaar, Ragnhild Van Der Straeten, and Dirk Deridder. 2010. Module superimposition: a composition technique for rule-based model transformation languages. *Software & Systems Modeling* 9, 3 (2010), 285–309.
- [16] Robert Waszkowski. 2019. Low-code platform for automating business processes in manufacturing. *IFAC-PapersOnLine* 52, 10 (2019), 376–381.