



OPEN ACCESS

EDITED BY

Kyriakos Kritikos,
University of the Aegean, Greece

REVIEWED BY

Kevin Lano,
King's College London, United Kingdom
Leandro Buss Becker,
Federal University of Santa Catarina, Brazil

*CORRESPONDENCE

Hamza Abdelmalek
✉ h.abdelmalek@edu.umi.ac.ma

RECEIVED 18 December 2023

ACCEPTED 28 May 2024

PUBLISHED 20 June 2024

CITATION

Abdelmalek H, Khriiss I and Jakimi A (2024)
Towards an effective approach for
composition of model transformations.
Front. Comput. Sci. 6:1357845.
doi: 10.3389/fcomp.2024.1357845

COPYRIGHT

© 2024 Abdelmalek, Khriiss and Jakimi. This is an open-access article distributed under the terms of the [Creative Commons Attribution License \(CC BY\)](https://creativecommons.org/licenses/by/4.0/). The use, distribution or reproduction in other forums is permitted, provided the original author(s) and the copyright owner(s) are credited and that the original publication in this journal is cited, in accordance with accepted academic practice. No use, distribution or reproduction is permitted which does not comply with these terms.

Towards an effective approach for composition of model transformations

Hamza Abdelmalek^{1*}, Ismaïl Khriiss² and Abdeslam Jakimi¹

¹GLISI Team, Faculty of Sciences and Technics, Moulay Ismail University, Errachidia, Morocco,

²Département de Mathématiques, d'Informatique et de Génie, Université du Québec à Rimouski, Rimouski, QC, Canada

Model Driven Engineering (MDE) adoption in the industry suffers from many technical and non-technical problems. One of the significant technical problems lies in the difficulty of building complex transformations from the composition of small and reusable transformations. Another problem resides in developing transformations from scratch in case they are missing. In this paper, we present an approach to how to handle these issues. The approach allows composing reusable transformations to build more complex ones by providing a catalog of prebuilt transformations targeting common architectures, frameworks, and design patterns. To give guidance and simplify the task of developing new transformations, we describe a platform description model of an entire system or a part of it in two views: a UML profile and a set of transformations. We also present three transformation types, each of which handles different abstraction design concerns. Generic transformations are small and reusable to build complex transformations, system-independent transformations are reusable and implement high-level design decisions, and system-specific transformations are not reusable and implement all design decisions needed for a given system. The approach is implemented as a plugin for a UML modeling tool and validated by developing a system that simulates the behavior of a gas station through model transformations built from the composition of reusable transformations.

KEYWORDS

model driven engineering, model transformation, transformations composition, reusable transformations, code generation

1 Introduction

Software development processes have evolved in response to business changes and customer needs. The goal of these processes is to deliver software on time and reduce resources, which is beneficial to individuals and organizations. Over the years, several software development processes have been used, such as Rapid Application Development (RAD) and Agile development, which focus on delivering software products as fast as possible. One such approach is Model-Driven Engineering (MDE), which supports minimum interaction with the code by abstracting the software development process using models. Low-code and no-code platforms are other software development environments with some commonalities with MDE (Di Ruscio et al., 2022). These platforms allow users with limited programming knowledge to develop their software products.

Model-driven architecture (MDA), proposed by the Object Management Group (OMG), is an implementation of MDE that provides a set of standards guiding the software development process using a set of models (Miller and Mukerji, 2003). Its two main models are platform-independent (PIM) and platform-specific (PSM). The PIM presents the system from the problem domain, while the PSM presents the system from the solution domain. OMG proposes another model called the Platform Description Model (PDM) used to describe a given architecture or platform. The software development process in MDA suggests applying a platform described in a PDM to the PIM to generate the PSM for a given system. The latter model is used to generate the system's source code.

Our MDE approach describes the PDM in two views: a UML profile and a set of transformations (Chénard et al., 2010). The UML profile is used to parameterize the PIM with the design decisions of a platform or architecture. Then, the transformations are applied to the parameterized PIM to generate the system's PSM or source code. We can use several PDMs to create the system in complex software products. Each PDM targets specific design decisions. We can simultaneously parametrize a PIM with multiple UML profiles using UML modeling software. However, it is challenging to develop and apply a set of transformations in the same parametrized PIM and generate a system that implements the design decisions of the applied PDMs. Hence, a technique is needed to ease the development and composition of model transformations.

For MDE to be applicable on a large scale, its evolution must follow the same trajectory as that of programming since it encounters the same challenges, such as tooling support, handling complexity, and developing quality products. High-level programming languages like C++ and Java emerged to replace low-level ones, simplifying programming. As software size increased, the necessity to organize and structure complex software became apparent, employing several reuse techniques like routines, class libraries, and frameworks. Throughout this evolution, the emphasis on software quality remained essential, which drove the development of comprehensive environments such as Visual Studio and NetBeans, which facilitate software development, debugging, testing, and deployment.

MDE also witnessed some attempts to propose solutions to these same challenges. Initially, practitioners relied on manual and *ad hoc* approaches for model transformation. However, with OMG's introduction of the MDA, the standardization of model transformation development became crucial. The OMG initiated an effort to develop the Query/View/Transformation (QVT) standard (OMG, 2009a). Subsequently, advanced model transformation languages like the Atlas Transformation Language (ATL) (Jouault et al., 2008) emerged to address the complexity of developing transformations. Some domains' complexity and their metamodels lead to complex transformations, which raises new challenges related to model transformations' portability, reusability, and maintainability. To overcome these challenges, practitioners and researchers proposed various techniques to facilitate the development of transformations and their reusability. We find the proposition of model transformation design patterns (Lano et al., 2018) to solve the most recurring problems in the field. One pattern that deals with transformation reusability is the transformation chain pattern (Lano et al., 2018), which addresses how to compose multiple transformations to build complex systems. Regarding tooling support, some proposals propose environments providing essential tools for debugging, testing, and integrating

transformations. Moreover, some environments enable the generation of sophisticated and high-quality systems by leveraging the composition of model transformations (Alvarez and Casallas, 2013; Basciani et al., 2018).

This paper presents our contribution to the adoption of MDE by simplifying the development of transformations and their reusability. Our approach allows the creation of complex systems by composing transformations implementing multiple PDMs, each targeting specific design decisions. In this sense, we introduce two types of transformations: generic and design transformations. Generic transformations are reusable and simple, reused to build more complex ones. Design transformations are divided into two types of transformations: system-independent and system-specific transformations. System-independent transformations (SIT) are constructed by composing generic transformations. They are reusable and can be complex, such as implementing a clean architecture, or simple transformations, such as those implementing an Observer pattern (Gamma et al., 1995). Recall that clean architecture (Martin, 2017) is a layered architecture for modern software development based on domain-driven design (DDD) (Evans, 2004) and best design principles. It allows the creation of systems that are independent of implementation technology. This independence is achieved by decoupling the business logic from the infrastructure implementation. System-specific transformations (SST) are always complex and support several design decisions. They can be developed from scratch or preferably be composed of SITs. The rationale behind these two design transformations is that two systems rarely make the same design decisions. The approach is implemented as a plugin for a UML modeling tool and validated by developing a system that simulates the behavior of a gas station through model transformations built from the composition of reusable transformations.

This paper is organized as follows. Section 2 explores the evolution of transformation languages and discusses related work in model transformation composition and its supporting tools. Section 3 describes our approach through a running example. Section 4 presents the tool supporting our approach. Section 5 concludes the paper and presents future work.

2 Related work

This section first explores the evolution of transformation languages and their applications. Next, we investigate the crucial aspect of reusability in transformation development and how it addresses the challenge of managing complexity. Finally, we investigate tools' role in enabling high-quality systems' development through model transformations.

Popular transformation languages are part of the Eclipse Modeling Framework (EMF) ecosystem.¹ For example, we find QVT (OMG, 2009a), ATL (Jouault et al., 2008), Henshin (Arendt et al., 2010), and Viatra (Balogh and Varró, 2006). QVT and ATL support declarative and imperative constructs for transformation development. Henshin and Viatra are graph-based transformation languages. According to Burgueno et al. it was found that practitioners in the industry tend to

¹ <https://www.eclipse.org/modeling/emf/>

favor the use of general-purpose programming languages such as Java for writing transformations (Burgueño et al., 2019). This preference stems from their familiarity with these languages and their desire to avoid the necessity of learning specialized model transformation languages. These dedicated languages often present a learning curve due to their functional nature, specifically tailored to address complex programming challenges (Höppner et al., 2022). Another good alternative for developing transformations is XSLT, which has a mature ecosystem with extensive tooling and community support. Another important aspect of XSLT is its portability, making it highly adaptable and easily integrated into various tools and environments. As it aligns with our specific needs, XSLT became our first choice as we sought a language that can be widely used in other aspects than model transformations.

Although not widely used, adopting model-based approaches and transformation languages has quickly seen the problem of managing complex transformations. Consequently, effectively managing the complexity of these transformations becomes crucial. In this regard, reuse is considered an important factor that addresses the challenges associated with the complexity of developing model transformations and their composition. Lano et al. classified the transformation chain as one of the important model transformation design patterns that address the issue (Lano et al., 2018). A transformation chain is also referred to as external transformation composition, where the output model of a transformation is used as input for the next transformation in the chain (Kleppe, 2006). Another technique is internal composition, where the definitions of multiple transformations are combined and then executed. In internal composition, transformations must be developed in the same transformation language, which is not a prerequisite in external composition.

Kusel et al. identified multiple reuse techniques in the field of model transformation (Kusel et al., 2013) that range in terms of reuse granularity from reusing parts of a transformation (Wagelaar et al., 2010) to reusing the whole transformation (Sen et al., 2012) and even composing multiple small model transformation chains (Yie et al., 2012).

Parts of transformations can be reused and composed with other parts using internal composition techniques to create transformations. The literature contains techniques such as rule inheritance (Wimmer et al., 2012a) and modularization (Kurtev et al., 2007).

Rule inheritance is similar to inheritance in object-oriented programming and allows specializing transformation rules from a base rule to avoid code duplication. This concept is supported by several model transformation languages (Wimmer et al., 2012a). Some support multiple inheritance, such as QVT, while others support single inheritance, such as ATL.

In modularization techniques, model transformation definitions are grouped into modules and reused in transformations. Modularization allows the execution of transformation definitions from multiple modules as a single transformation. We find techniques such as variability-based rule (Strüber et al., 2018), module superimposition (Wagelaar et al., 2010), factorization (Sánchez Cuadrado and García Molina, 2008), phases (Cuadrado and Molina, 2009), and many objective transformation modularization (Fleck et al., 2017). Strüber et al. introduced the variability-based rule, a representation that groups similar model transformation rules to avoid maintenance problems. Module superimposition, proposed by Wagelaar et al., is another technique that allows reusing transformation

definitions from different modules. Cuadrado and Molina proposed a factorization approach to extract common transformation definitions and compose them using phases, where phasing is a mechanism to organize model transformation definitions into modules or phases, thereby increasing their reusability and maintainability. Fleck et al. proposed an automatic approach to divide large ATL model transformations into reusable and smaller transformations.

In our approach, we have defined generic transformations (GTs), which are small, parameterized, and reusable transformations used to build more complex transformations using internal composition. The concept is similar to the concept of phases (Cuadrado and Molina, 2009) with differences such as their execution order. Phases are executed explicitly where the user specifies the execution order or implicitly where the transformation engine executes the phases according to their order in the transformation definition. Our approach differentiates between two types of GTs: containers and building blocks, where the container GT specifies the execution order of the building block GTs.

Concerning reuse techniques with large granularity, we find in the literature approaches inspired by generic programming that introduce generic transformations (Sánchez Cuadrado et al., 2011; Sen et al., 2012; Wimmer et al., 2012b). They allow the creation of reusable transformations across similar source or target metamodels. These transformations map source to target concepts instead of concrete metamodel elements. Cuadrado et al. introduce generic transformations (Sánchez Cuadrado et al., 2011) and their binding to concrete metamodels, in addition to their composition, using a component model (Cuadrado et al., 2014) that allows the building of complex transformations. Wimmer et al. improved the work of Sánchez Cuadrado et al. (2011) by automatically adding adapters to transformations to address the problem of structural heterogeneity between metamodels (Wimmer et al., 2012b). Sen et al. take another approach to reuse model transformations by transforming the target metamodel to become a subset of the transformation's input metamodel (Sen et al., 2012).

In our approach, we have introduced system-independent transformations (SITs), reusable transformations used in constructing complex systems through their composition with other SITs. Each SIT implements partial or all design decisions of a PDM.

The effectiveness of model-driven approaches in delivering high-quality transformations and generating reliable systems depends significantly on the availability of robust tooling support (Bucchiarone et al., 2020). We find the propositions of environments and tools that leverage external composition to create complex systems in the literature. These environments facilitate the development process by suggesting, validating, or executing transformation chains. However, many of these environments are immature and lack appropriate testing and debugging tools to ensure the reliability and correctness of the transformations and generated systems.

Aranega et al. used feature models to guide transformation chains, where they validated their approach in an environment dedicated to embedded systems called Gaspard2 (Aranega et al., 2012). In this case study, they assisted end users by organizing a set of transformations as a feature model and proposing the appropriate chain of transformations based on the preferences of the end users. Alvarez and Casallas propose a tool called MTC flow that allows the design, development, and deployment of model transformation chains (Alvarez and Casallas, 2013). They evaluated the tool through

two final projects of a model-driven development (MDD) course, where students used the tool to implement their projects and conducted a survey about their experience with the tool. They reported that the tool is easy to use but can be enhanced by extending the documentation and improving the view of the graphical editor. FTG + PM is a framework proposed by Lucio et al. that provides a set of artifacts used in model transformation chains and their executions (Lúcio et al., 2013). The formalism transformation graph (FTG) defines the transformations, and the process model (PM) describes the chaining of those transformations. They presented the power window control software in automotive applications as a case study. ChainTracker is a tool that focuses more on traceability and presents the composition of rule-based model-to-model (M2M) transformation and template-based model-to-text (M2T) transformation (Guana and Stroulia, 2014). CITRIC is a tool that recommends to developers whether there is a multiple transformation chain to link a source model to a target metamodel using the shortest path algorithms (Basciani et al., 2018). If multiple chains are discovered, the optimal one is selected and executed based on two criteria: metamodel coverage and information loss. They validated their approach by transforming a sample-KM model into an XML specification. Wires (Rivera et al., 2009) is a domain-specific language (DSL) (Fowler, 2010) for orchestrating transformation models developed in ATL. UniTI is a tool proposed by Vandhoof et al. that facilitates the composition and execution of model transformations without knowing implementation details (Vanhooff et al., 2007). To illustrate their approach, they transformed a storage model into the corresponding Java code by producing a transformation chain from the following model transformations: a transformation that converts a storage model to a UML model, a transformation that modifies the associations within the UML model, and a transformation that transforms UML into equivalent Java models. Etien et al. presented localized transformations restricted to a specific transformation task (Etien et al., 2015). Combining those localized transformations using the “extend” operator in the case of incompatible metamodels allows the construction of large transformations. As a case study, they chained localized transformations to transform the UML profile of MARTE (OMG, 2009b) into implementation platform languages such as SystemC² and OpenMP.³

These approaches employ the composition at the transformation level, while we have adopted another approach by composing the results at the model level. In this sense, we have proposed a third type of transformation called system-specific transformation (SST). An SST is composed of multiple SITs depending on the system requirements. The execution of an SST leads to the execution of its containing SITs, where each SIT results in a partial PSM. Later, all the partial PSMs of the SST are combined using a merge tool. Our approach favors the separation of concerns, as each SIT transformation deals only with a specific design concern, and their composition does not require the definition and use of intermediate metamodels as in a transformation chain. Another important factor when developing transformations and their composition is the adoption of standards

and best practices; that’s why we provide a process for developing and reusing transformations according to the MDA approach.

Some approaches provide model transformation libraries or catalogs to avoid developing transformations from scratch. Wimmer et al. provided a library of mapping operators, which are transformations that map concepts from input to output metamodels (Wimmer et al., 2010). Wang et al. offer a library of object-oriented design patterns in XSLT (Wang et al., 2007). Aranega et al. suggest model transformation chains based on a library of transformations (Aranega et al., 2012). Etien et al. provided a library of localized transformations for developing embedded systems applications (Etien et al., 2015). Our approach presents a catalog of PDMs supporting design patterns or architectural patterns.

3 Description of approach

3.1 The Metamodel

The OMG introduced the Platform Model or the Platform Description Model (PDM) to describe an implementation platform. According to the OMG, a PDM “provides a set of technical concepts, representing the different kinds of parts that make up a platform and the services provided by that platform. It also provides, for use in a platform-specific model, concepts representing the different kinds of elements to be used in specifying the use of the platform by an application (Miller and Mukerji, 2003).” In another work, we defined the PDM in two views (see Figure 1a): the UML profile and a set of transformations (Chénard et al., 2010).

The UML profile allows the extension of the UML metamodel to support a given domain or platform using stereotypes and tagged values. A UML profile is expressed using concepts and constraints between them. Concepts are the main building blocks of an implementation platform. A concept is defined by its name, type (classifier (class or interface), attribute, operation, parameter, or artifact),⁴ description, and design concerns. A design concern is used to formulate a well-known design issue. It is defined by its name, a type (stereotype or tagged value), concerned UML elements (package, classifier, attribute, operation, parameter, generalization, association, association end, and dependency), and description. A constraint is used to maintain the integrity of the implementation platform by restricting the use of concepts. We define a constraint by its name, the concerned concepts, its type (dependency, compatibility, incompatibility, refinement), and a description.

A transformation applies to a specific model element type representing its context (see Figure 1b). The model element type is related to the elements of the UML metamodel. It can be a package, classifier, attribute, operation, parameter, generalization, association, association end, or dependency. The context represents the condition of applying a transformation based on the properties of a model element type. For example, the context may be the existence of a stereotype property named Repository in a class model element.

² <https://systemc.org>

³ <https://www.openmp.org>

⁴ Note that our approach is currently limited to the types of class diagram elements. However, it is easily extendable to support other types of elements in a model.

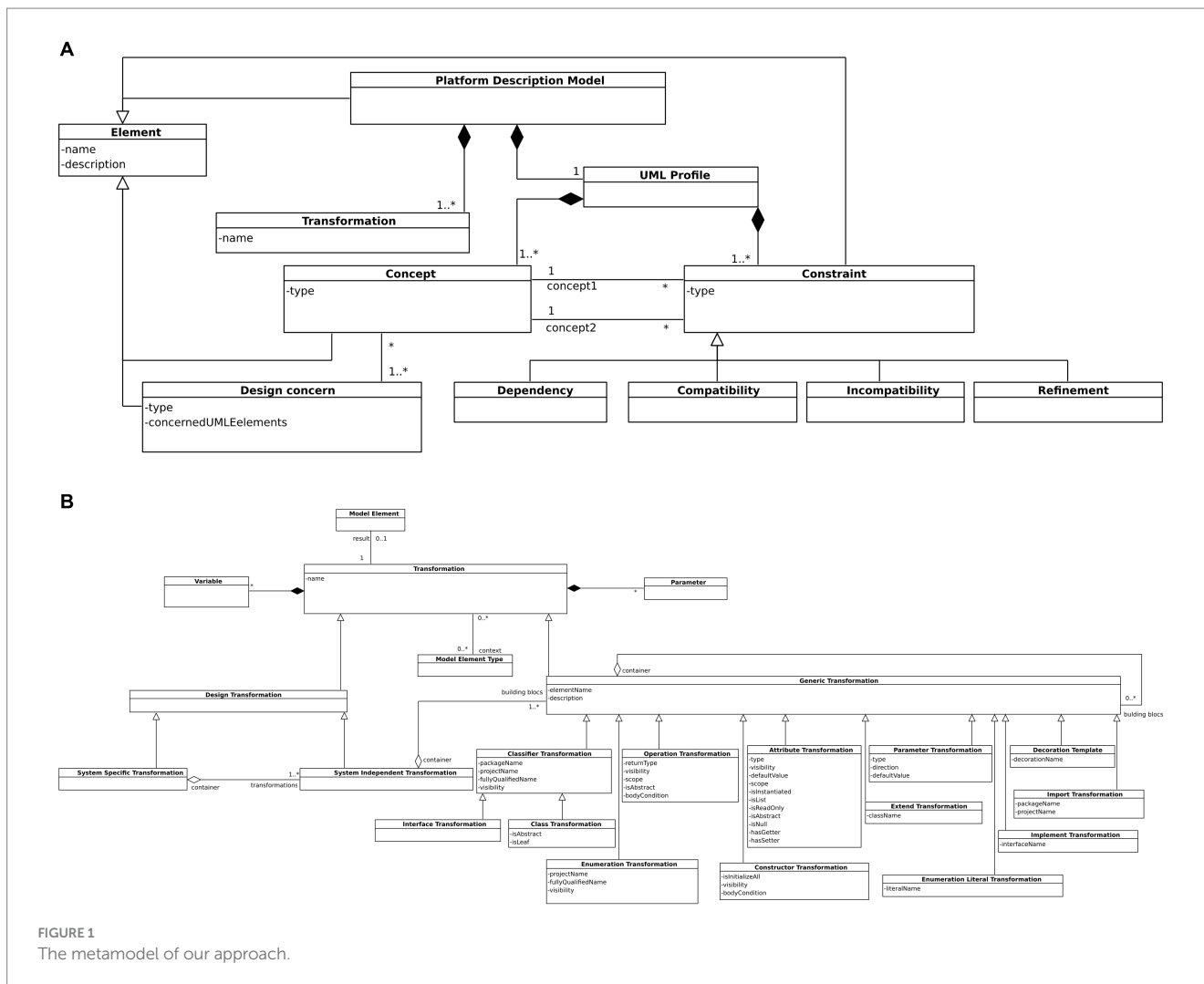


FIGURE 1 The metamodel of our approach.

Performing a transformation results in a model element that implements an architecture or platform concept. The resulting model element can be a classifier (class or interface), operation, attribute, parameter, or artifact.

A transformation is identified by its name and can have parameters. The existence of parameters depends on the type of transformation. As mentioned before, a transformation can be a generic or design transformation. Design transformations can, in turn, be system-independent or system-specific.

Generic transformations are parameterized transformations reused as building blocks to construct system-independent transformations using internal composition. A generic transformation (GT) can be a container, a building block, or both. An example of a container GT is the class transformation, which reuses other generic transformations such as the operation and constructor GTs. The latter two GTs can play both roles because they are considered containers for the parameter-building block GT.

System-independent transformation (SIT) is constructed by defining its context and reusing generic transformations as building blocks. An SIT can be applied directly to a model or reused with other SITs to build more complex systems. We can distinguish two types of SITs: complex SITs that implement design decisions of a main architecture, such as MVC or clean architecture, and simple

SITs that implement small design designs, such as design patterns. The latter SITs must contain a parameter that specifies their container architecture or project.

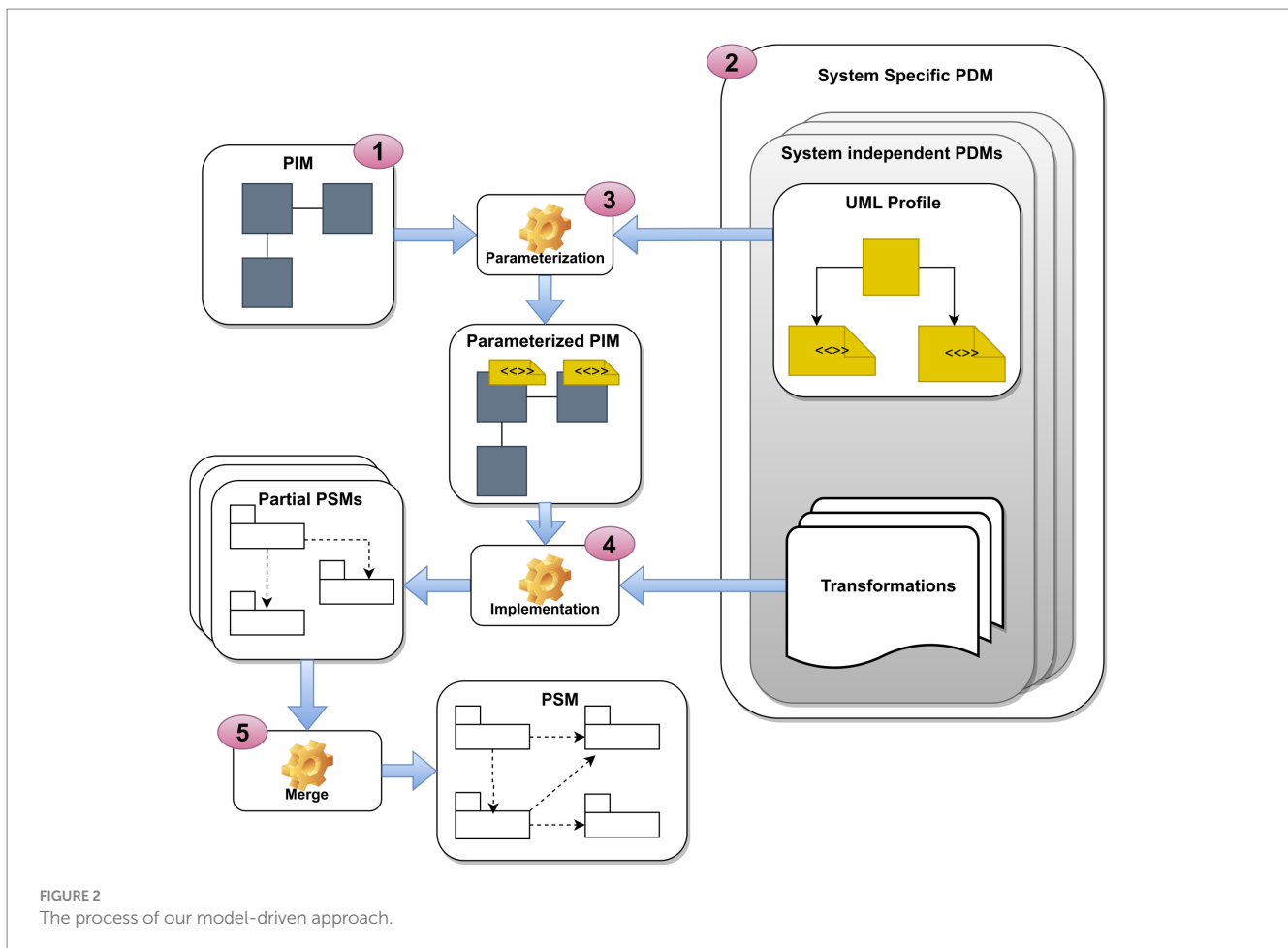
System-specific transformation (SST) is another type of design transformation. It is constructed by reusing SITs to implement design decisions of a system. Unlike other transformations, SSTs are not reusable and are built for a specific system with a particular combination of design decisions.

In this sense, A PDM can be either be system-independent or system-specific. A system-specific PDM may be developed from scratch or composed by reusing system-independent PDMs.

3.2 Overview of approach

Figure 2 presents an overview of our approach to generating the source code of a system from its model. It consists of five steps:

- 1 Modeling the problem domain of the software system (PIM).
- 2 Specifying the software system's implementation platform (system-specific PDM). This specification should result from reusing existing system-independent PDMs instead of creating the specification from scratch.



- 3 Parametrizing the PIM using the UML profile of the system-independent PDMs.
- 4 Applying the transformations of the system-independent PDMs on the parameterized PIM. This application results in partial PSMs.
- 5 Merging the partial PSMs using a merge tool.

To illustrate our approach, we use a system that simulates the behavior of a gas station as a running example. The PIM of the system contains 16 classes and two enumerations (see Figure 3). The pump class manages its components: tank, gun, display, meter, and motor. Employees can supervise the pump's status and the tank's level and change the gas price. The diagram also shows classes supporting accounts, transactions, and payments of customers. The station contains three pumps that a customer can use when authorized by an employee.

The development of the gas station system requires several major design decisions, including the choice of the reference architecture and the technologies to be used. It will also require several small, localized design decisions on certain system parts. We chose clean architecture as the system's main architecture.

The architecture is organized into four packages (see Figure 4)⁵: the SharedKernel package defines common classes and interfaces

shared between systems; the Core package represents the business logic of the system; the Infrastructure package includes the implementation of the repositories and the interactions with the data sources and third-party libraries; and the Presentation package represents the user interface of the system.

The system needs to track the pump's status and notify other components, such as its display. We, therefore, decided to use the Observer design pattern for this task. We used the following technologies: DotNET 6, C# programming languages, SQL Server database, and Entity Framework for object-relational mapping (ORM). Hence, the system will reuse at least the transformations of the clean architecture and the Observer pattern.

The first step in constructing a PDM is to define its UML profile. The clean architecture UML profile contains, for instance, Entity, Repository, and Service concerns. The UML profile of the observer design pattern includes concerns such as Observer, Subject, and Notify. The next step is to define the transformations of each PDM. These transformations are system-independent and built using generic transformations. For example, the clean architecture PDM contains the Entity SIT, which implements the Entity concern. Figure 5 shows an excerpt of the building blocks of this SIT using the generic transformations: Import, Extend, and Class. The figure also shows the context of the SIT, which is UML elements of type Class with the stereotype Entity.

The SITs from the clean architecture and design pattern PDMs are reused to construct an SST that generates the PSM of the system.

⁵ Our implementation of the clean architecture in DotNET was inspired from the Ardalis Github repository (<https://github.com/ardalis/cleanarchitecture>).

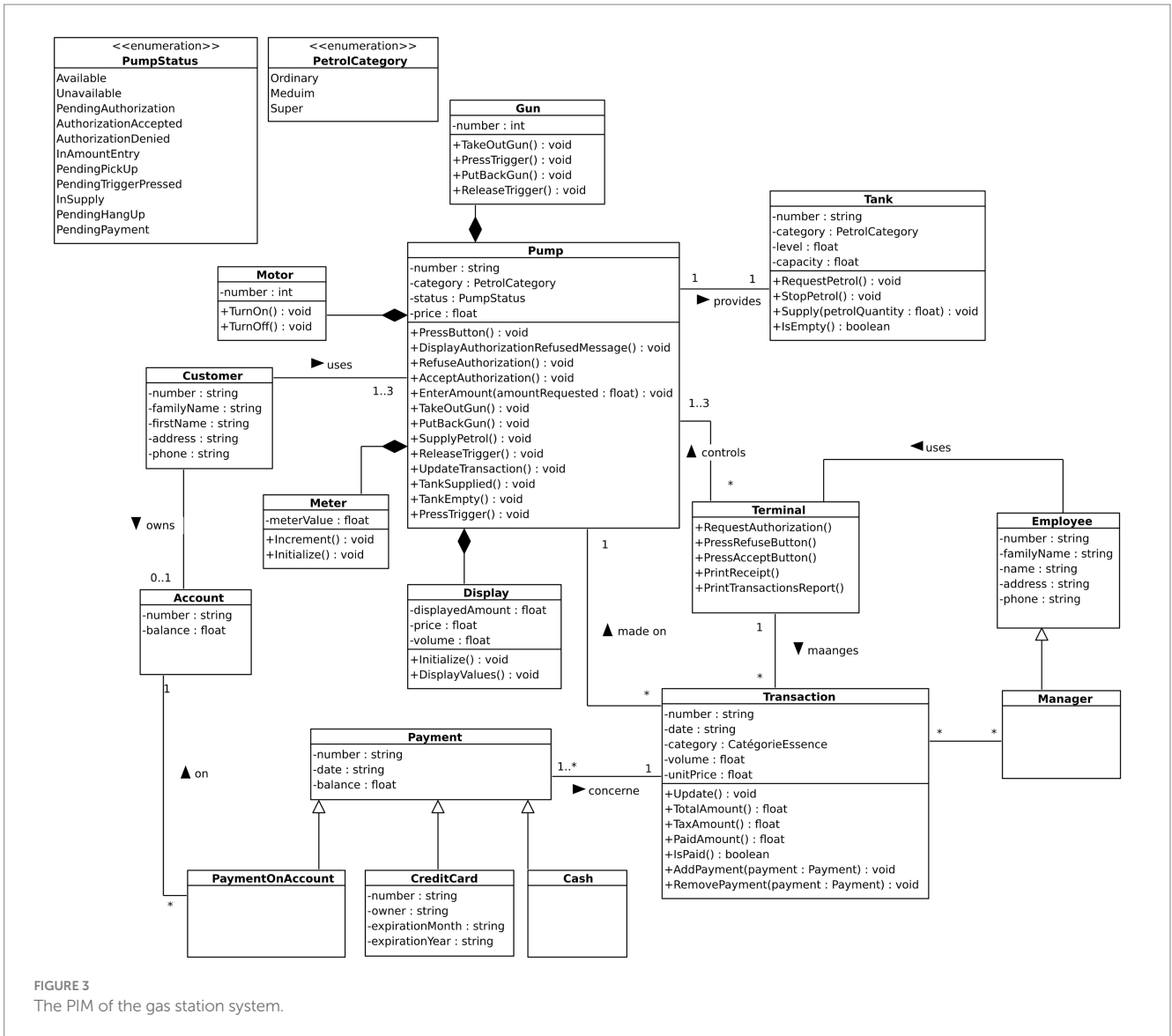


FIGURE 3 The PIM of the gas station system.

After the specification of the PDMs, we parameterize the PIM with different design decisions using the UML profile of the PDMs. The parameterization of the pump, motor, and meter classes is presented in Figure 6. For example, the pump class is parameterized with the following stereotypes Entity, AggregateRoot, Repository, and Service of the clean architecture PDM and Subject of the observer PDM. Next, we apply the SST to the parameterized PIM, which executes the SITs of the PDMs. The partial PSMs created by applying SITs are merged into a single PSM representing the complete system.

The final step is to apply an M2T transformation to the PSM to generate the system's source code. An excerpt of the PSM of the system is presented in Figure 7, showing only the Core layer of the clean architecture. It is organized into five sub-packages: the subpackage Entities contains domain entities; the subpackage DesignPatterns includes the implementation of the design patterns; the subpackage Interfaces contains the definition of services and repositories of the system; the subpackage Services includes the implementation of the services; and the subpackage Specifications

contains the implementation of the Specification design pattern in DDD (Evans, 2004). A specification contains the criteria necessary for validation or retrieving an entity. An example of a specification is searching for a pump by its category.

4 Tool support

4.1 Approach implementation

To implement our MDE approach, we have developed a plugin for Visual Paradigm⁶ (VP) using its open API and the Java programming language. By extending VP's functionalities, the plugin allows the specification of PDMs, PIM parameterization, and code generation. The PDM specification step is simplified by providing a set of user

⁶ <https://www.visual-paradigm.com/>

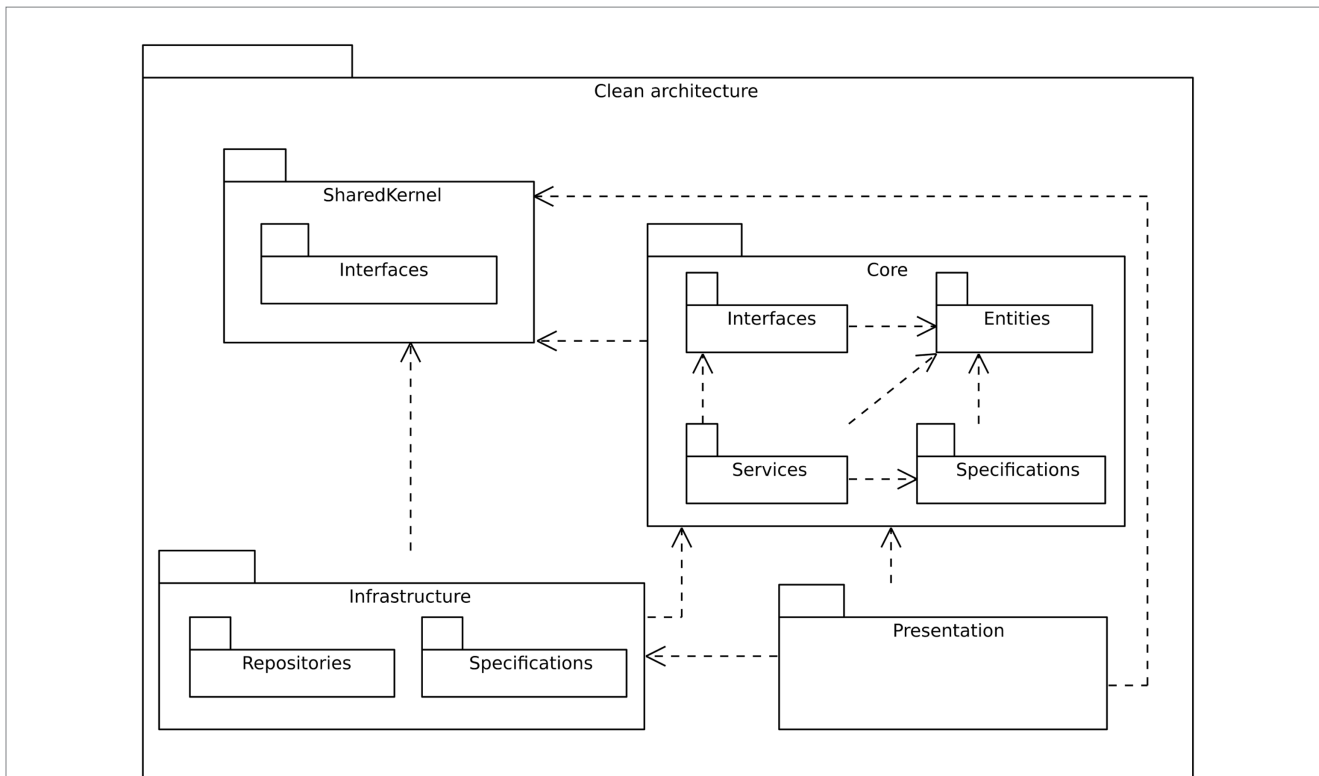


FIGURE 4 The structure of the clean architecture.

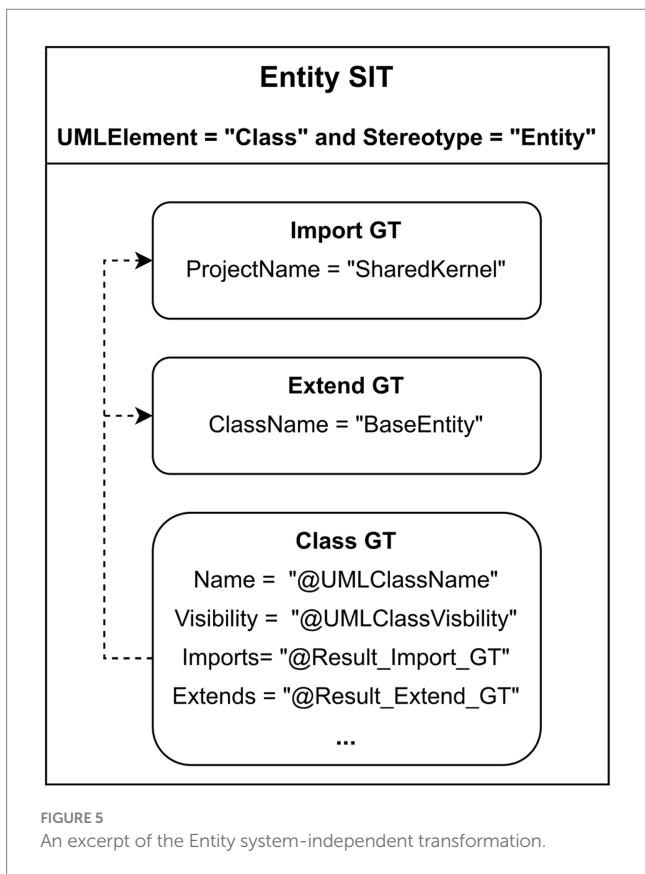


FIGURE 5 An excerpt of the Entity system-independent transformation.

interfaces where a user can create the UML profile and specify the transformations.

With the VP modeling editor, PIM parameterization becomes easier, where the user can select the concerned UML elements and the necessary PDMs, and then apply various design decisions. Finally, it allows code generation using the parameterized PIM and the defined transformations. Moreover, the tool employs XML format for importing and exporting the specified PDMs and the parameterized PIM. The plugin's architecture is illustrated in Figure 8 and is organized into four packages: *Controllers*, *UserInterface*, *Structures*, and *Utilities*, in addition to the XML file `plugin.xml` and the class `MainMDE`.

The XML file `plugin.xml` is crucial in defining the plugin by providing essential information such as an identifier, a description, the provider, the main class, action sets, and context-sensitive action sets. The main class (`MainMDE` in Figure 8) implements the interface `com.vp.plugin.VPPlugin`, and serves as the first executed class when the plugin is loaded. Action sets and context-sensitive action sets allow the customization of toolbars and menus in VP. We can differentiate between two types of actions: those defined on the main or diagram toolbars using action sets and those defined in the popup menu within the diagram editor using context-sensitive action sets. In the plugin, we have defined two actions on the main toolbar for the PDM definition and code generation and one in the popup menu for the PIM parameterization step, which involves selecting the concerned UML elements from the class diagram. Each action is associated with a class presenting the action controller that implements either

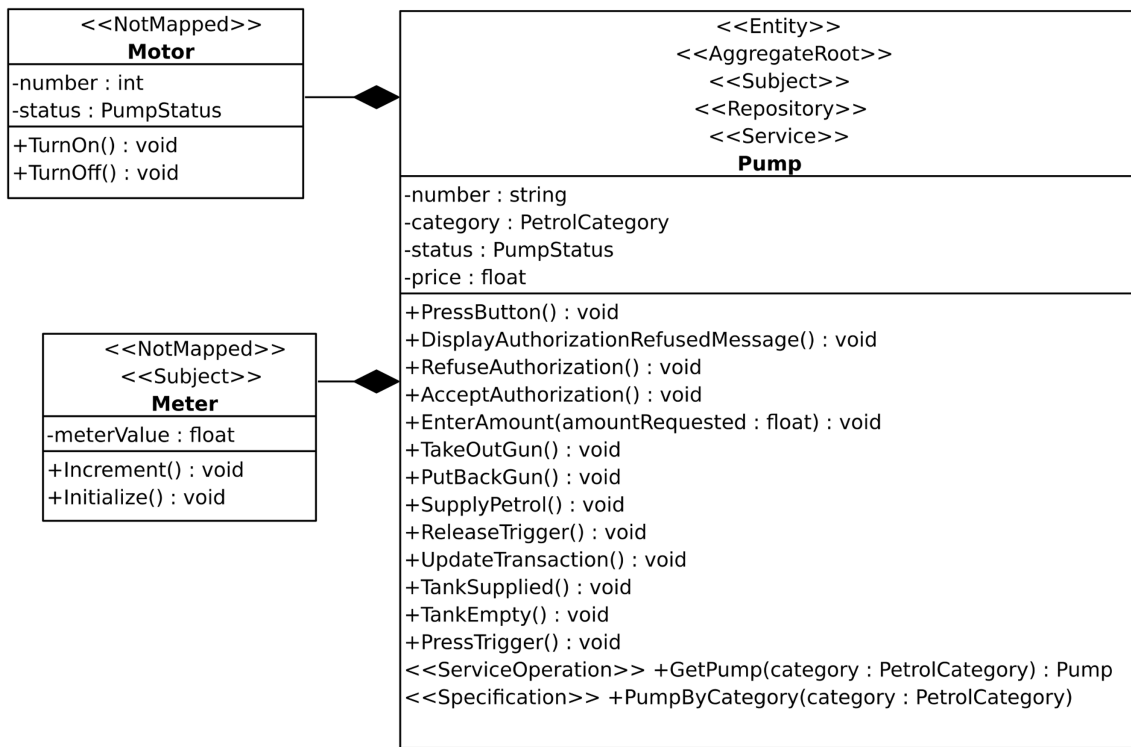


FIGURE 6 An excerpt of the PIM parameterization.

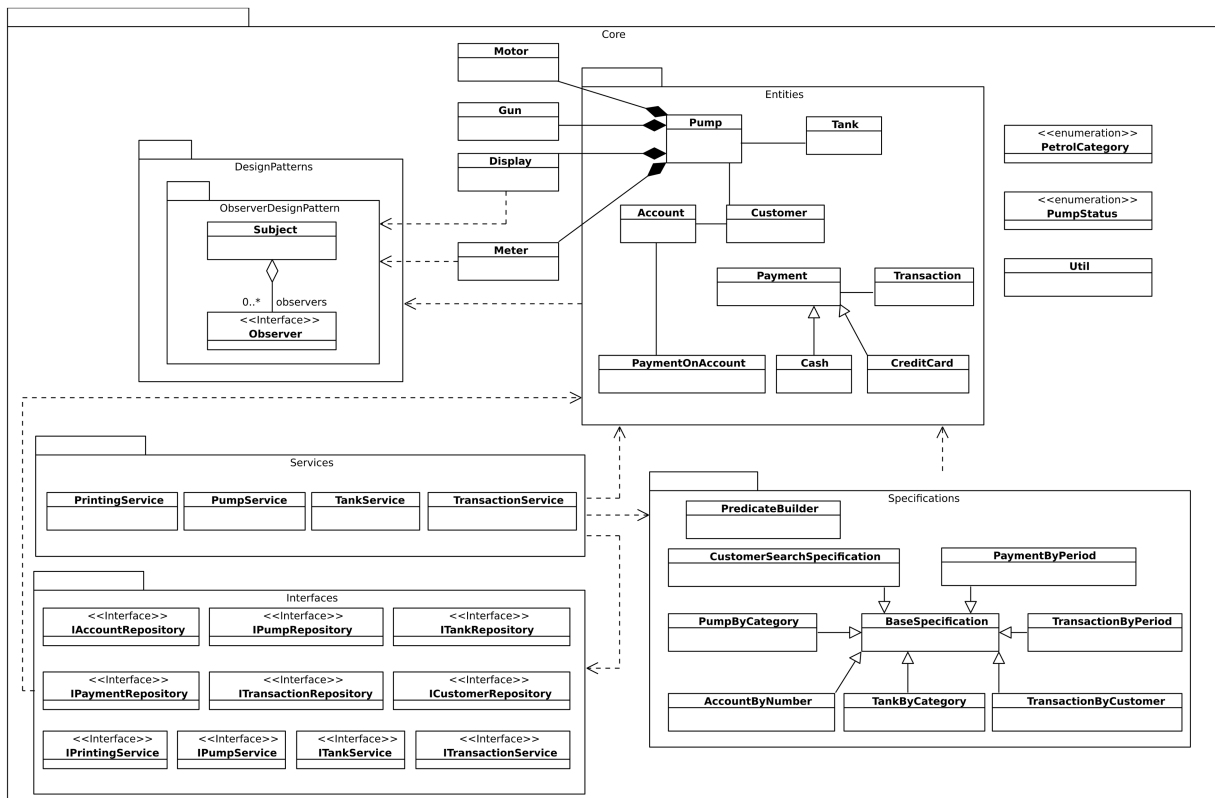


FIGURE 7 An excerpt of the PSM of the system.

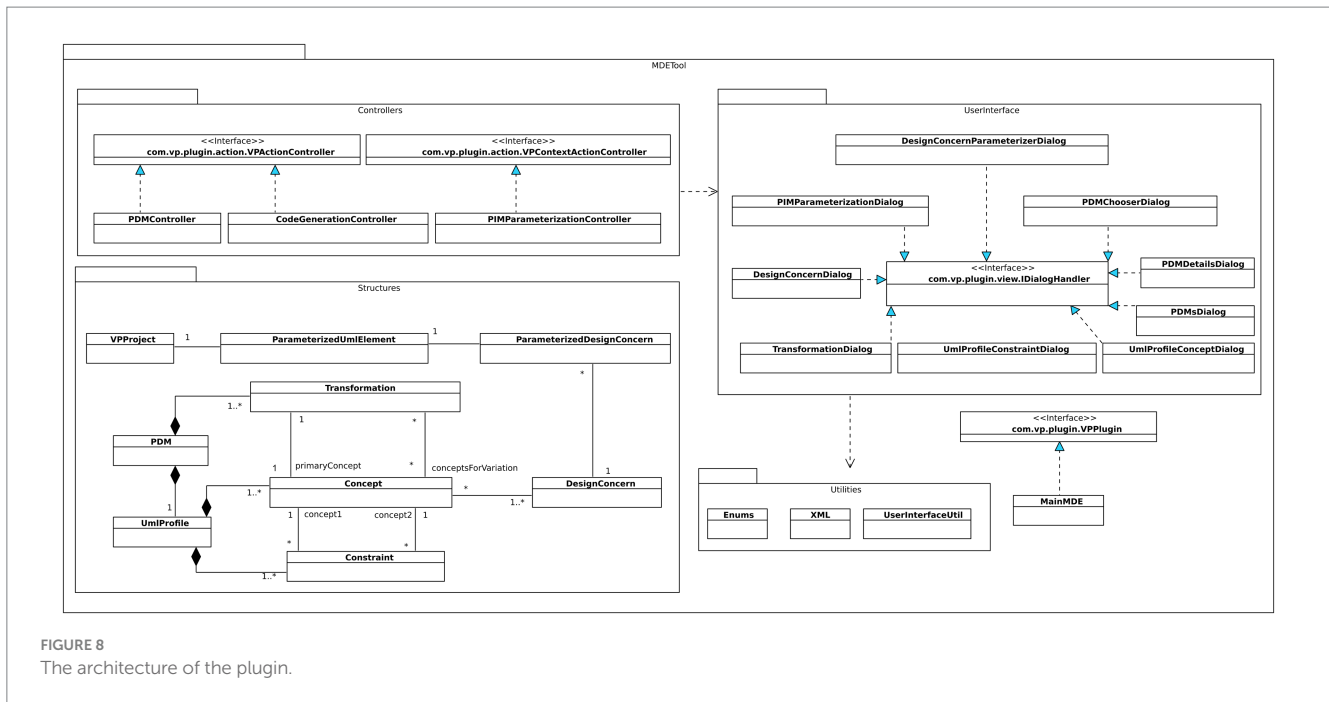


FIGURE 8 The architecture of the plugin.

the interface `VPActionController` for toolbar actions or the interface `VPContextActionController` for popup menu actions. These action controllers are implemented in the package `Controllers`, which contains the classes `PDMController`, `CodeGenerationController`, and `PIMParameterizationController`.

The package `UserInterface` consists of dialogs designed to facilitate interactions with the tool. Each dialog within the package is required to implement the interface `com.vp.plugin.view.IDialogHandler` from the open API. In the plugin, we differentiate the dialogs into two types: main dialogs and sub-dialogs. Main dialogs are triggered by executing an action controller, whereas sub-dialogs are invoked through other dialogs. The package includes two main dialogs: `PDMsDialog` and `PIMParameterizationDialog`. `PDMsDialog` is triggered by the execution of the action controller `PDMController`, while the action controller `PIMParameterizationController` triggers `PIMParameterizationDialog`. `PDMsDialog` is responsible for adding, editing, removing, and saving PDMs. When adding or editing a PDM, the sub-dialog `PDMDetailsDialog` is invoked (see Figure 9), allowing the user to add, edit, or remove concepts along with their corresponding design concerns, constraints, or transformations. These tasks are performed using a set of sub-dialogs, namely `UmlProfileConceptDialog`, `DesignConcernDialog`, `UmlProfileConstraintDialog`, and `TransformationDialog`. `PIMParameterizationDialog` simplifies the process of PIM parameterization. It allows users to select the required PDMs using the sub-dialog `PDMChooserDialog` and listing the chosen UML elements as a tree. Users have the flexibility to select multiple PDMs for parameterizing the PIM, resulting in the creation of individual trees within the dialog. Each tree has the PDM name as its root and the selected UML elements as its children. In Figure 10, the dialog `PIMParameterizationDialog` displays two trees corresponding to the PDMs of the clean architecture and the observer design pattern. Once the trees are displayed, users can proceed to parameterize each UML element. This parameterization is achieved by applying the necessary

stereotypes and tagged values with the assistance of `DesignConcernParameterizerDialog`.

The package `Structures` contains the classes presented in the metamodel, such as `PDM`, `UMLProfile`, `Concept`, `DesignConcern`, `Constraint`, and `Transformation`. In addition, it includes the classes `ParameterizedDesignConcern`, `ParameterizedUmlElement`, and `VPProject` related to the parameterization process. The class `ParameterizedDesignConcern` captures the design concern with the value given during the parameterization. The class `ParameterizedUmlElement` encapsulates the UML element with its parameterized design concerns. The class `VPProject` captures the parameterized UML elements in a given class diagram.

The package `Utilities` contains helper classes like `XML`, `UserInterfaceUtil`, and `Enums`. The class `XML` contains methods to import and export the PDM and parametrized PIM in an XML format. The class `UserInterfaceUtil` includes utilities that simplify controls' disposition and files and folders management. The `Enums` class has a list of enumerations used in the plugin; we find the following enumerations: `TransformationType`, `DesignConcernType`, `UMLElementType`, `UMLProfileConceptType`, and `UMLProfileConstraintType`.

The choice of a transformation language is an important factor when developing transformations. In our MDE approach, we have chosen XSLT due to its portability, as various development environments support it. Those environments provide features for debugging and testing XSLT transformations.

Figure 11 gives an excerpt of an SIT using XSLT. It describes the Entity SIT within the clean architecture PDM. This transformation is executed when a class is parameterized with stereotype `Entity`, as indicated in the match attribute. As a result, it creates a set of classes that extend the `BaseEntity` class. To achieve this, the SIT reuses two generic transformations: `Class` and `Extend` GTs. To this end, this SIT imports and calls the `extend` GT (see Figure 12), specifying `BaseEntity` as the value for the `class_name` parameter. The result of this transformation is stored in a variable, which is reused in the

FIGURE 9
The user interface for specifying a PDM specification.

Class GT alongside other parameters like the class name and visibility. By utilizing this approach, the Entity SIT effectively ensures that the classes parameterized with the Entity stereotype extend the required BaseEntity class, enabling conformance to the clean architecture principles. For simplicity, this example only shows an excerpt from the Entity SIT. In reality, this SIT consists of additional blocks created by attribute, operation, parameter, constructor, and import GTs. The attribute GT is responsible for generating the required attributes of a given class, by providing their properties such as name, type, default value, and visibility. The parameter GT is a building block transformation reused in the operation and constructor GT. The Import GT is also reused to import the package of the extended class BaseEntity.

The source code of the VP plugin and the model transformation composition technique are available on the GitHub repositories (Abdelmalek et al., 2023a,b), respectively.

4.2 Comparison with alternative transformation language

In this subsection, we discuss implementing our approach using the QVT transformation language. The OMG introduced QVT as the

standard language for defining model transformations within the MDA framework (OMG, 2009a). QVT supports both declarative and imperative styles of transformation definitions through its QVT Relations (QVTr) and QVT Operational (QVTo) sublanguages, respectively. We opted for QVTo to implement our approach due to its comprehensive capabilities for specifying transformations, which include using the Object Constraint Language (OCL) and imperative constructs such as conditions and loops. By choosing QVTo, we aim to provide a detailed evaluation of its effectiveness relative to our proposed MDA methodology and to illustrate how our approach can be applied using other model transformation languages beyond XSLT.

4.2.1 QVT implementation

As previously outlined, our approach facilitates model transformation reusability by defining and composing three types of transformations: GTs, SITs, and SSTs. In the subsequent paragraphs, we assess the effectiveness of the QVTo language in implementing these distinct transformation categories.

GTs are parameterized transformations that are essential for constructing more complex transformations. In QVTo, these parameterized transformations can be created through mappings or helpers within a specific library, enabling the explicit declaration of parameter names and their types. When a complex

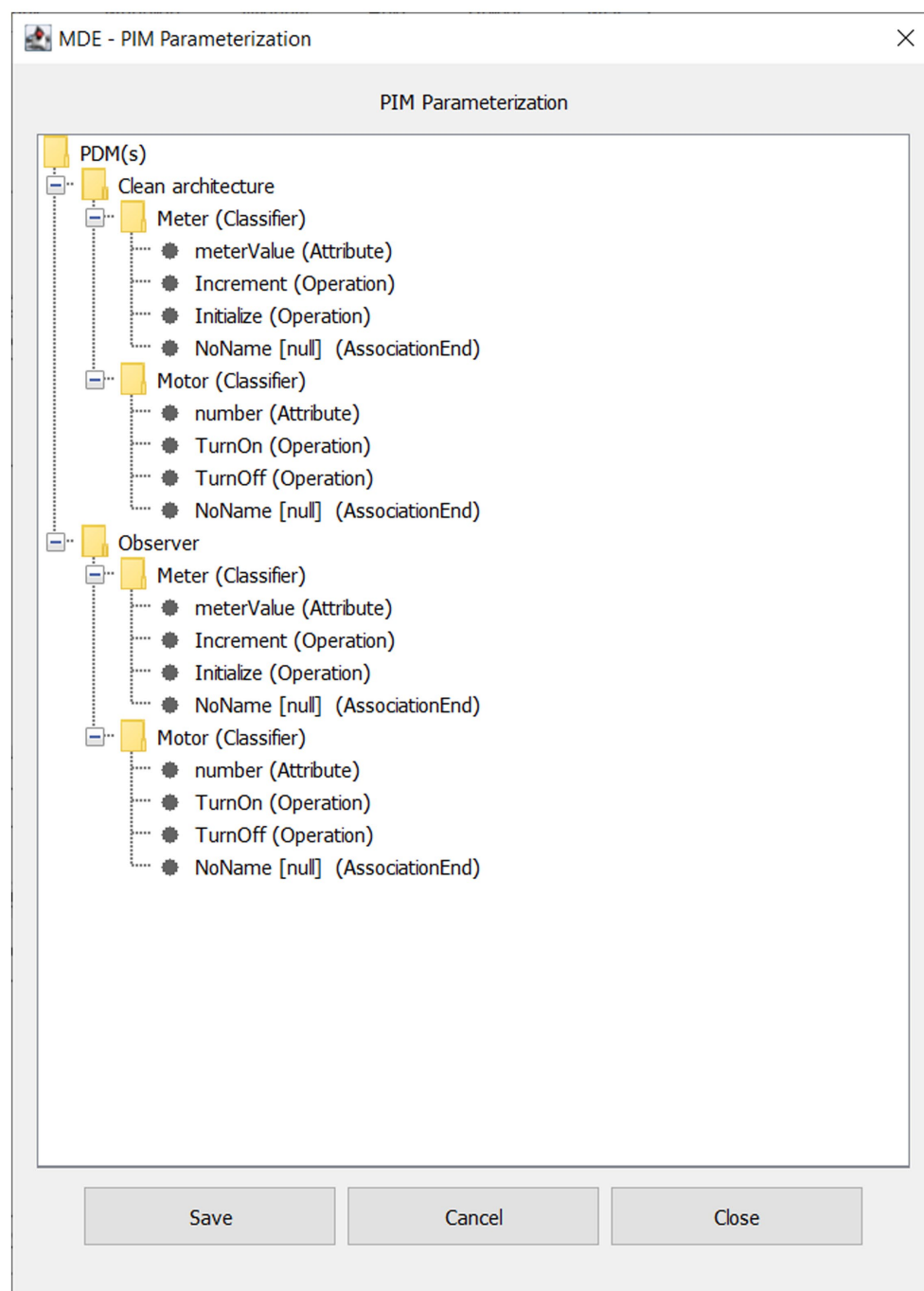


FIGURE 10
The user interface for parameterizing a PIM.

transformation employs a GT, it is required to specify the values for its parameters. For example, in defining a GT intended for creating attributes of classes, parameters such as the attribute's name (of type String) and its visibility (of type VisibilityKind) must be explicitly defined. An excerpt of this attribute GT is illustrated in Figure 13, represented as a mapping transformation within the generic_transformations library. In contrast to XSLT, QVTo offers the distinct advantage of allowing for the explicit specification of parameter types.

SITs reuse GTs to implement specific concepts within a PDM. QVTo facilitates the reuse of transformations by allowing the import of mappings and helpers from other libraries and modules. The condition for applying a SIT can be specified using a when clause preceding a mapping, enabling targeted selection of UML elements. Moreover, QVTo includes queries that extract data from models, similar to XPath in XSLT. These queries assist in retrieving data for SITs and supplying it to the parameters of the GTs when they are reused. An excerpt of the singleton SIT implemented in QVTo is

```

<xsl:template match="//Class[Stereotypes/Stereotype[@Name = 'Entity']]" mode="entity_template">
  <xsl:variable name="entity_extend">
    <xsl:call-template name="extend_template">
      <xsl:with-param name="class_name">BaseEntity</xsl:with-param>
    </xsl:call-template>
  </xsl:variable>
  <xsl:call-template name="class_template">
    <xsl:with-param name="class_visibility">
      <xsl:value-of select="@Visibility"/>
    </xsl:with-param>
    <xsl:with-param name="class_name">
      <xsl:value-of select="@Name"/>
    </xsl:with-param>
    <xsl:with-param name="class_extends">
      <xsl:copy-of select="exsl:node-set($entity_extend)/node()"/>
    </xsl:with-param>
  </xsl:call-template>
</xsl:template>

```

FIGURE 11
An excerpt of the Entity system-independent transformation.

```

<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:output method="xml" omit-xml-declaration="no" indent="yes" encoding="UTF-8"/>
  <xsl:template name="extend_template">
    <xsl:param name="class_name"/>
    <xsl:element name="Extend">
      <xsl:value-of select="$class_name"/>
    </xsl:element>
  </xsl:template>
</xsl:stylesheet>

```

FIGURE 12
The extend generic transformation.

illustrated in Figure 14, targeting UML classes parameterized with the Singleton stereotype. This SIT reuses the attribute GT by importing the library containing generic transformations.

SSTs compose multiple SITs to implement various design decisions tailored to a particular system. SSTs function as the primary entry point for executing these transformations, containing the main function that triggers the entire process. Furthermore, SSTs import multiple libraries to efficiently reuse the SITs that align with the system's specific requirements. Consequently, each PDM is encapsulated within its own library, which includes mappings pertinent to its unique concepts. Figure 15 illustrates an SST that combines an SIT of the clean architecture with an SIT designed to implement the Singleton design pattern.

In summary, we have demonstrated how our approach can be implemented using the standard QVTo language. QVTo's imperative nature enhances reusability and composition in model transformations. It incorporates key features such as parameterized rules, modularization, and conditional execution. These features are crucial for facilitating the creation of GTs, SITs, and SSTs, thereby optimizing the efficiency of the transformation process and improving its maintainability.

4.2.2 Comparative analysis

In this subsection, we conduct a comparative analysis of our methodology implemented using both XSLT and QVTo, focusing on four key criteria: transformation definition, transformation


```

library generic_transformations;

modeltype UML uses uml("http://www.eclipse.org/uml2/3.0.0/UML");

mapping Class :: attribute_GT(attName : String, attType : Type, attVisibility:
VisibilityKind, isAttStatic: Boolean) : Property {
  name := attName;
  type := attType;
  isStatic := isAttStatic;
  visibility:= attVisibility
}

```

FIGURE 13
An excerpt of the attribute GT using QVTo.

```

library singleton_PDM;

import generic_transformations;

modeltype UML uses uml("http://www.eclipse.org/uml2/3.0.0/UML");

mapping Class :: instance_SIT() : Class
  when {self.getAppliedStereotypes()->any(name = "Singleton")<>null}
{
  name := self.name;
  ownedAttribute += self.map generic_transformations::attribute_GT("instance",
self, VisibilityKind::private, true);
}

```

FIGURE 14
An excerpt of the singleton SIT using QVTo.

```

import clean_architecture_PDM;
import singleton_PDM;

modeltype UML uses uml("http://www.eclipse.org/uml2/3.0.0/UML");

transformation clean_singleton_transform(in source: UML, out target: UML);

main() {
  source.rootObjects()[Model].map Model2Model();
}

mapping Model :: Model2Model(): Model {
  name := self.name;
  packagedElement += self.packagedElement[Class].map map_SITs();
}

mapping Class :: map_SITs()
{
  self.map clean_architecture_PDM::entity_SIT();
  self.map singleton_PDM::instance_SIT();
}

```

FIGURE 15
An SST that composes clean architecture entity and singleton.

reusability, code generation efficiency, and development setup and integration.

In terms of transformation definition, XSLT employs a template-based approach where the transformation logic is defined through templates that match specific elements or patterns in the input model. This method provides considerable flexibility and expressiveness in defining transformations. However, managing and organizing the transformation logic for complex transformations can be challenging. To overcome these difficulties, we have developed three distinct types of transformations. Each type addresses specific concerns and is further integrated using an internal composition technique. In contrast, QVT adopts an imperative approach, where transformation logic is explicitly defined through mappings between the input and output models. This method facilitates a more structured and straightforward definition of transformations. Furthermore, while XSLT is primarily optimized for M2T transformations, QVT is specifically designed for M2M transformations (Willink, 2018).

Transformation reusability is a critical aspect of model transformation. Our methodology underscores the importance of reusing transformations from libraries and composing them to construct complex software systems. We employ XSLT to segment transformation logic into reusable templates, enabling developers to efficiently reuse transformation components across multiple scenarios. The familiarity and widespread adoption of XSLT within the software engineering community enhance its value in our methodology. Consequently, developers can leverage existing knowledge and resources to expedite software development by reusing transformations created by others. In contrast, QVT offers a more structured approach to transformation reusability and composition. Its modular architecture, support for parameterized rules, and library definitions significantly aid in the composition of transformations. This structured approach allows developers with a background in MDE to assemble complex transformations from reusable building blocks, providing a robust framework for systematic model transformation.

In terms of code generation efficiency, XSLT excels particularly in simpler transformations. Our methodology enhances XSLT's capabilities, enabling the creation of more complex transformations by composing smaller, reusable elements to generate advanced software systems. Additionally, the familiarity and user-friendliness of XSLT make it accessible to a broad spectrum of developers. This accessibility promotes collaborative development and facilitates knowledge sharing within the community, further leveraging the collective expertise. On the other hand, while QVT is primarily tailored for M2M transformations, rather than M2T transformations, it can still be effectively utilized for code generation. This is achieved through its integration with other M2T tools such as Acceleo.⁷

Development setup and integration assess the ease with which the transformation development environment can be established and how seamlessly transformations can be integrated with other tools. XSLT demonstrates high portability and ease of integration, requiring minimal dependencies. Its compatibility across various platforms and environments ensures better interoperability with different tools, simplifying the setup process. XSLT typically only requires a software engine to execute transformations, making it a straightforward choice

for many developers. Additionally, its widespread adoption and extensive documentation support facilitate easy configuration. On the other hand, setting up QVT might involve more complexities due to its lower usage and more specialized requirements. Although QVT is supported by some Integrated Development Environments (IDEs) and modeling tools, such as the Eclipse IDE, additional configurations and plugins may be necessary. This can add layers of complexity to the initial setup process, requiring a deeper understanding and more time to achieve optimal integration and functionality.

In conclusion, while QVT provides robust support for transformation composition and boasts advantages in terms of expressiveness and performance, XSLT remains a vital element of our methodology. Its flexibility, expressiveness, and widespread familiarity render it indispensable for implementing our approach. By strategically leveraging the capabilities of XSLT alongside QVT, developers are equipped to create robust and scalable transformation solutions that meet the requirements of complex software projects. Recognizing the unique benefits of each, some approaches, such as those proposed by Li et al. (2011), have attempted to synergistically combine the strengths of both XSLT and QVT.

4.2.3 Benefits of our approach

Implementing our methodology using XSLT offers significant flexibility and expressiveness, granting precise control over transformation processes. This adaptability is crucial for handling complex scenarios through the composition of reusable transformations. For example, our approach introduces the definition of SITs by composing GTs, which is then followed by the construction of SSTs through the further composition of SITs. By structuring transformation logic into reusable templates and libraries, developers can accelerate the development process and ensure consistency across various transformation scenarios. This strategy is enhanced by developing a VP plugin, which further supports the reuse of transformations and complete PDMs from a library, amplifying the efficiency and scalability of the development workflow.

While our approach yields significant benefits when implemented using XSLT, there are numerous opportunities for enhancements that could further augment its effectiveness. Developing additional tooling support tailored for XSLT-based transformations, including IDEs with syntax highlighting, code completion, and advanced debugging functionalities, could considerably streamline the development process and enhance developer productivity. Moreover, integrating XSLT-based transformations with modern technologies and frameworks, such as cloud computing platforms, could broaden their applicability across various domains. Such integration would not only ensure the continued relevance of XSLT in meeting the dynamic needs of MDE but could also extend its utility to sophisticated applications like model transformation composition within low-code platforms, as discussed by Sahay et al. (2020).

As demonstrated in subsection 3.3.1, our methodology for model transformation reusability is designed to be language-independent, a fact validated by its implementation in QVT. However, despite its strengths, QVT faces challenges that may limit its broader appeal. The limited tooling support and steep learning curve associated with QVT make it less accessible for developers outside the MDE community. These factors can restrict its adoption compared to XSLT, which is generally more familiar and supported within the broader software engineering field.

⁷ <https://eclipse.dev/acceleo/>

5 Conclusion and future work

This paper presented an approach that allows composing reusable transformations to build more complex ones by providing a catalog of prebuilt transformations targeting common architectures, frameworks, and design patterns. To give guidance and simplify the task of developing new transformations, we described a platform description model in two views: a UML profile and a set of transformations. We also introduced three transformation types, each handling different abstraction design concerns. Generic transformations are small and reusable to build complex transformations, system-independent transformations are reusable and implement high-level design decisions, and system-specific transformations are not reusable and implement all design decisions needed for a given system. The approach is implemented as a plugin for a UML modeling tool and validated by developing a system that simulates the behavior of a gas station through model transformations built from the composition of reusable transformations.

In future work, we plan to enrich our catalog of prebuilt transformations and better integrate the results of one of our previous works into our plugin, allowing an interactive discovery of the platform description models of legacy systems.

Data availability statement

Publicly available source code of the tools can be found at: <https://github.com/AHamza14/MDE-tool> and <https://github.com/AHamza14/Model-transformation-composition>.

References

- Abdelmalek, H., Khriiss, I., and Jakimi, A. (2023a). MDE tool [Online]. Available at: <https://github.com/AHamza14/MDE-tool> (Accessed 2023).
- Abdelmalek, H., Khriiss, I., and Jakimi, A. (2023b). Model transformation composition technique [Online]. Available at: <https://github.com/AHamza14/Model-transformation-composition> (Accessed 2023).
- Alvarez, C., and Casallas, R. (2013). "MTC flow: a tool to design, develop and deploy model transformation chains". In Proceedings of the workshop on ACadeMics Tooling with Eclipse.
- Aranega, V., Etien, A., and Mosser, S. (2012). "Using feature model to build model transformation chains". In Model Driven Engineering Languages and Systems: Proceedings of the 15th International Conference, MODELS 2012 Innsbruck, Austria, Springer, September 30–October 5, 2012.
- Arendt, T., Biermann, E., Jurack, S., Krause, C., and Taentzer, G. (2010). "Henshin: advanced concepts and tools for in-place EMF model transformations". In Model Driven Engineering Languages and Systems: Proceedings, Part I 13th International Conference, MODELS 2010 Oslo, Norway, Springer, October 3–8, 2010.
- Balogh, A., and Varró, D. (2006). "Advanced model transformation language constructs in the VIATRA2 framework". In Proceedings of the 2006 ACM symposium on Applied computing.
- Basciani, F., Di Ruscio, D., D'Emidio, M., Frigioni, D., Pierantonio, A., and Iovino, L. (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.
- Bucchiarone, A., Cabot, J., Paige, R. F., and Pierantonio, A. (2020). Grand challenges in model-driven engineering: an analysis of the state of the research. *Softw. Syst. Model.* 19, 5–13. doi: 10.1007/s10270-019-00773-6
- Burgueño, L., Cabot, J., and Gérard, S. (2019). The future of model transformation languages: an open community. *J. Object Technol.* 18, 1–11. doi: 10.5381/jot.2019.18.3.a7
- Chénard, G., Khriiss, I., and Salah, A. (2010). "Towards the discovery of implementation platform description models of legacy object-oriented systems". In Workshop on Processes for Software Evolution and Maintenance (WoPSEM 2010) IEEE.
- Cuadrado, J. S., Guerra, E., and de Lara, J. (2014). A component model for model transformations. *IEEE Trans. Softw. Eng.* 40, 1042–1060. doi: 10.1109/TSE.2014.2339852
- Cuadrado, J. S., and Molina, J. G. (2009). Modularization of model transformations through a phasing mechanism. *Softw. Syst. Model.* 8, 325–345. doi: 10.1007/s10270-008-0093-0
- Di Ruscio, D., Kolovos, D., de Lara, J., Pierantonio, A., Tisi, M., and Wimmer, M. (2022). Low-code development and model-driven engineering: two sides of the same coin? *Softw. Syst. Model.* 21, 437–446. doi: 10.1007/s10270-021-00970-2
- Etien, A., Muller, A., Legrand, T., and Paige, R. F. (2015). Localized model transformations for building large-scale transformations. *Softw. Syst. Model.* 14, 1189–1213. doi: 10.1007/s10270-013-0379-8
- Evans, E. (2004). *Domain-driven design: Tackling complexity in the heart of software*. Boston, MA, USA: Addison-Wesley Professional.
- Fleck, M., Troya, J., Kessentini, M., Wimmer, M., and Alkhazi, B. (2017). Model transformation modularization as a many-objective optimization problem. *IEEE Trans. Softw. Eng.* 43, 1009–1032. doi: 10.1109/TSE.2017.2654255
- Fowler, M. (2010). *Domain-specific languages*. Boston, MA, USA: Pearson Education.
- Gamma, E., Johnson, R., Helm, R., Johnson, R. E., and Vlissides, J. (1995). *Design patterns: Elements of reusable object-oriented software*. Munchen, Germany: Pearson Deutschland GmbH.
- Guana, V., and Stroulia, E. (2014). "Chaintracker, a model-transformation trace analysis tool for code-generation environments". In Theory and Practice of Model Transformations: Proceedings of the 7th International Conference, ICMT 2014, Held as Part of STAF 2014, York, UK, Springer, July 21–22, 2014.
- Höppner, S., Haas, Y., Tichy, M., and Juhnke, K. (2022). Advantages and disadvantages of (dedicated) model transformation languages: a qualitative interview study. *Empir. Softw. Eng.* 27:159. doi: 10.1007/s10664-022-10194-7
- Jouault, F., Allilaire, F., Bézivin, J., and Kurtev, I. (2008). ATL: a model transformation tool. *Sci. Comput. Program.* 72, 31–39. doi: 10.1016/j.scico.2007.08.002

Author contributions

HA: Writing – original draft, Writing – review & editing, Conceptualization, Data curation, Formal analysis, Investigation, Resources, Software, Validation, Visualization. IK: Conceptualization, Formal analysis, Investigation, Methodology, Project administration, Supervision, Writing – review & editing, Resources, Validation. AJ: Writing – review & editing.

Funding

The author(s) declare that no financial support was received for the research, authorship, and/or publication of this article.

Conflict of interest

The authors declare that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.

Publisher's note

All claims expressed in this article are solely those of the authors and do not necessarily represent those of their affiliated organizations, or those of the publisher, the editors and the reviewers. Any product that may be evaluated in this article, or claim that may be made by its manufacturer, is not guaranteed or endorsed by the publisher.

- Kleppe, A. (2006). First European Workshop on Composition of Model Transformations-CMT 2006.
- Kurtev, I., van den Berg, K., and Jouault, F. (2007). Rule-based modularization in model transformation languages illustrated with ATL. *Sci. Comput. Program.* 68, 138–154. doi: 10.1016/j.scico.2007.05.006
- Kusel, A., Schönböck, J., Wimmer, M., Kappel, G., Retschitzegger, W., and Schwinger, W. (2013). Reuse in model-to-model transformation languages: are we there yet? *Softw. Syst. Model.* 14, 537–572. doi: 10.1007/s10270-013-0343-7
- Lano, K., Kolahdouz-Rahimi, S., Yassipour-Tehrani, S., and Sharbaf, M. (2018). A survey of model transformation design patterns in practice. *J. Syst. Softw.* 140, 48–73. doi: 10.1016/j.jss.2018.03.001
- Li, D., Li, X., and Stolz, V. (2011). QVT-based model transformation using XSLT. *ACM SIGSOFT Softw. Eng. Notes* 36, 1–8. doi: 10.1145/1921532.1921563
- Lúcio, L., Mustafiz, S., Denil, J., Vangheluwe, H., and Jukss, M. (2013). "FTG+ PM: an integrated framework for investigating model transformation chains". In *SDL 2013: Model-Driven Dependability Engineering: Proceedings of the 16th International SDL Forum 16: Montreal, Canada, Springer, June 26–28, 2013*, 182–202.
- Martin, R. (2017). *Clean architecture: A craftsman's guide to software structure and design*. Hoboken, NJ, USA: Prentice Hall.
- Miller, J., and Mukerji, J. (2003). *MDA guide version 1.0. 1: Object management group Inc.*
- OMG, Q. (2009a). Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification.
- OMG, U. (2009b). Profile for MARTE: Modeling and analysis of real-time embedded systems specification, version 1.0.
- Rivera, J. E., Ruiz-Gonzalez, D., Lopez-Romero, F., Bautista, J., and Vallecillo, A. (2009). Orchestrating ATL model transformations. *Proc. MtATL* 9, 34–46.
- Sahay, A., Indamutsa, A., Di Ruscio, D., and Pierantonio, A. (2020). "Supporting the understanding and comparison of low-code development platforms". In *2020 46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*.
- Sánchez Cuadrado, J., and García Molina, J. (2008). "Approaches for model transformation reuse: factorization and composition". In *Theory and Practice of Model Transformations: Proceedings of the First International Conference, ICMT 2008, Zürich, Switzerland, Springer, July 1–2, 2008*.
- Sánchez Cuadrado, J., Guerra, E., and De Lara, J. (2011). "Generic model transformations: write once, reuse everywhere". In *Theory and Practice of Model Transformations: Proceedings of the 4th International Conference, ICMT 2011, Zurich, Switzerland, June 27–28, 2011*.
- Sen, S., Moha, N., Mahé, V., Barais, O., Baudry, B., and Jézéquel, J.-M. (2012). Reusable model transformations. *Softw. Syst. Model.* 11, 111–125. doi: 10.1007/s10270-010-0181-9
- Strüber, D., Rubin, J., Arendt, T., Chechik, M., Taentzer, G., and Plöger, J. (2018). Variability-based model transformation: formal foundation and application. *Form. Asp. Comput.* 30, 133–162. doi: 10.1007/s00165-017-0441-3
- Vanhooff, B., Ayed, D., Van Baelen, S., Joosen, W., and Berbers, Y. (2007). Uniti: a unified transformation infrastructure. In *Model Driven Engineering Languages and Systems: Proceedings of the 10th International Conference, MoDELS 2007, Nashville, USA, Springer, September 30–October 5, 2007*.
- Wagelaar, D., Van Der Straeten, R., and Deridder, D. (2010). Module superimposition: a composition technique for rule-based model transformation languages. *Softw. Syst. Model.* 9, 285–309. doi: 10.1007/s10270-009-0134-3
- Wang, X.-B., Wu, Q.-Y., Wang, H.-M., and Shi, D.-X. (2007). "Research and implementation of design pattern-oriented model transformation". In *2007 International Multi-Conference on Computing in the Global Information Technology (ICCGI'07): IEEE*.
- Willink, E.D. (2018). "A text model-use your favourite M2M for M2T". In *MoDELS (Workshops)*, 89–102.
- Wimmer, M., Kappel, G., Kusel, A., Retschitzegger, W., Schönböck, J., and Schwinger, W. (2010). "Surviving the heterogeneity jungle with composite mapping operators". In *Theory and Practice of Model Transformations: Proceedings of the Third International Conference, ICMT 2010, Malaga, Spain, Springer, June 28–July 2, 2010*.
- Wimmer, M., Kappel, G., Kusel, A., Retschitzegger, W., Schönböck, J., Schwinger, W., et al. (2012a). Surveying rule inheritance in model-to-model transformation languages. *J. Obj. Technol.* 11, 31–46. doi: 10.5381/jot.2012.11.2.a3
- Wimmer, M., Kusel, A., Retschitzegger, W., Schönböck, J., Schwinger, W., Cuadrado, J. S., et al. (2012b). Reusing model transformations across heterogeneous metamodels. *Electr. Commun. EASST.* 50, 1–13. doi: 10.14279/tuj.eceasst.50.722.795
- Yie, A., Casallas, R., Deridder, D., and Wagelaar, D. (2012). Realizing model transformation chain interoperability. *Softw. Syst. Model.* 11, 55–75. doi: 10.1007/s10270-010-0179-3