ABSTRACT
Model Transformation is the key aspect of Model Driven Engineering (MDE), a model-centric software development approach aiming at improving the quality and productivity of software development processes. This paper presents a metamodel based model transformation framework, which is based on a formal modeling notation – Z-based Object-Oriented Modeling notation (ZOOM). It includes a set of supporting tools aiming at delivering the benefits in practical applications of model driven engineering. The potential benefits of the proposed model transformation framework include: 1) readability and rigorousness of meta-model definitions; 2) simplicity of transformation definition; and 3) extensibility of transformation templates. The architecture and design of the framework is discussed and comparisons with related research work are provided to show the benefits of this framework.

KEYWORDS
Model Driven Engineering, Modeling, Metamodeling, Model Transformation

1 Introduction
Model Driven Engineering (MDE) tackles the elusive problem of system development by promoting the usage of models as the primary artifact to be constructed and maintained [1, 2]. MDE shifts software development from a code-centric activity to a model-centric activity. Accomplishing this shift entails developing support for modeling concepts at different levels of abstraction and transforming abstract models to more concrete descriptions of software. In another word, MDE reduces complexity in software development through modularization and abstraction [3].

Because of MDE’s potential to dramatically change the way we develop applications, companies are already working to deliver supporting technologies [4]. However, there is no universally accepted definition of the requirements for a MDE infrastructure and many requirements for MDE support are unclear or even unspecified. Notwithstanding the lack of standards, with careful reading of related researches [5], we argue that the main issues MDE infrastructure is facing are: (1) incompleteness in existing modeling notations; (2) lack of effective model transformation mechanism. Considering above issues, we provide an alternate solution, which is a new model-driven engineering framework including a formal modeling notation and a set of supporting tools aiming at the realization of the benefits of true model-driven engineering. We have developed a new formal modeling notation called Z-based Object-Oriented Modeling notation or ZOOM, which is based on the formal specification notation Z [6–8], and several key components of UML-2. ZOOM is a simple, precise, and easy to use modeling language. It has dual representation textual and visual. The syntax of textual representation is defined precisely in EBNF grammar. The formalism of modeling notation provides a solid foundation for metamodeling which is an important factor in model transformation. With a simplified metamodeling design, we are able to develop an extensible model transformation process.

In summary, our work objective is to apply such an overall research approach in realization of MDE focusing on model transformation. The notation and metamodel design lays the foundation of my tool development and the tool development in turn demonstrates the validity and advantage of the design.

This paper is organized as follows: Section 2 presents the ZOOM architecture and notation. Then section 3 covers the characteristics of our model transformation approach and the model transformation process. Section 4 discusses related research work and compares them with our approach and finally Section 5 concludes the paper.
2 ZOOM Architecture and Notation

While UML-2 is widely used as a visual modeling language to support MDE, it has several weaknesses. UML-2 is not specifically designed for MDE, so its models are generally informative without providing definitive views. Also, while UML-2 provides multiple visual views to present similar aspects of a software design, it lacks an inherent mechanism to enforce consistency between these views. Additionally, UML-2 lacks user and system interface design notations. To overcome these obstacles, we propose to enhance the UML-2 models and meta-model to include support for formal syntax and semantics and to provide a new UI model. The result is a new formal notation called ZOOM. ZOOM stands for Z-based Object-Oriented Modeling. It is based on the formal specification notation Z [7–9], which is in turn based upon set theory and mathematical logic. ZOOM provides a human-readable syntax to specify the mathematical model in Z. A complete description of how ZOOM supports Z notation can be found in [10].

Figure 1 shows the overall architecture of the ZOOM models for a software system. The functional requirements derive the structural, behavioral and UI models. ZOOM provides a pre-defined event model, which is processed by an event-driven framework, to bind the structural, behavioral, and UI models together [11]. The integrated ZOOM model will be processed by the Knowledge-based Model Compilation Tools resulting in different implementations of the software system based on the specific platform and knowledge base.

3 Metamodel based Transformation Approach

Figure 2 shows the basic structure of our model transformation framework. A transformation engine takes the Hierarchical Relational Metamodel(HRM) defined source model as input, and use a template comprise of a set of transformation rules to produce output model in a format specified by the templates. In another word, the output from the transformation engine is a transformation of the input model. We will discuss the characteristics of our approach in the following subsections.

3.1 Source Model Representation

We use ZOOM notation to represent Platform Independent Model(PIM). ZOOM notation has a textual syntax defined by BNF, which gives us a simplified way to define and use ZOOM metamodel. Since ZOOM provides the textual syntax in a form that most programming languages have, we are able to build an internal representation of ZOOM models in a structure similar to Abstract Syntax Tree(AST), only the node in the tree will be constructs of the modeling language instead of constructs of programming language. However, to capture more complicated modeling language constructs like association. We adopt mathematical collection to depict the relationships of different constructs. Considering it’s tree structure and such relationships, we name our metamodel Hierarchical Relational Metamodel(HRM).
3.2 A Metamodelling Language

Metamodelling is a critical part of our transformation approach. It provide a mechanism to unambiguously define modeling languages - ZOOM in our case. It is the prerequisite for a model transformation tool to access and make use of the models. We will now look into the design of our Hierarchical Relational Metamodel (HRM).

3.2.1 Hierarchical Relational Metamodel

The fact that ZOOM notation has a textual syntax defined by BNF gives us a simplified way to define and use ZOOM model's metamodel. From implementation point of view, metamodel defines the internal representation of models. In programming language, this internal representation often takes the form of Abstract Syntax Tree (AST) that can be processed by interpreter or compiler [12,13]. Since ZOOM provides the textual syntax in a form that most programming languages have, we are able to build an internal representation of ZOOM models in a structure similar to Abstract Syntax Tree. The only difference is the nodes in the tree are constructs of the modeling language instead of constructs of programming language. To capture more complicated modeling language constructs like association, we also adapt mathematics collection to depict the relationships of different constructs.

3.2.2 HRM Definition

We provide the following definition of HRM:

**Definition 3.1.** Hierarchical Relational Metamodel is a 3-tuple: HRM = (N, C, R)

N is a set of nodes: \( N = \{ n_1, n_2, ..., n_j \} \)

C is a relation on \( N \times N \), which forms a tree structure that has one root and no unconnected nodes. Each node may have zero or more children. In another word, a node is either a leaf (i.e. with no children) or can be decomposed as one or more children and each child forms a subtree

\[ R = \{ r_1, r_2, ..., r_k \} \]

is a set of relations between nodes, where \( r_i \) is a relation on \( N \times N \).

Figure 3 shows a simple class diagram that has four classes: Student, Graduate, Undergraduate and Course. The corresponding HRM diagram is also show in Figure 3 in the middle. This metamodel can be represented as \( (N, C, R) \) according to Definition 3.1. More specifically, we can elaborate the contents of its three components as:

<table>
<thead>
<tr>
<th>HRM component</th>
<th>content</th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
<td>{ ClassDiagram, Student, Graduate, enroll, x, y, Student.name, Graduate.advisor, ... }</td>
</tr>
<tr>
<td>C</td>
<td>{ (Student, Student.name),(Graduate, Graduate.advisor), (Graduate, Graduate.thesis), ... }</td>
</tr>
<tr>
<td>R</td>
<td>{ superClass, subClass, leftAssociationEnd, rightAssociationEnd }</td>
</tr>
<tr>
<td>superClass</td>
<td>{ (x, Student), (y, Student) }</td>
</tr>
<tr>
<td>subClass</td>
<td>{ (x, Graduate), (y, Undergraduate) }</td>
</tr>
<tr>
<td>leftAssociationEnd</td>
<td>{ (enroll, Student) }</td>
</tr>
<tr>
<td>rightAssociationEnd</td>
<td>{ (enroll, Course) }</td>
</tr>
</tbody>
</table>

The components superClass, subClass, leftAssociationEnd and rightAssociationEnd are relations between classes Student, Graduate, Undergraduate, Course and relationship enroll, x, y.
3.3 Transformation Template

The rule set shown in Figure 2 is a collection of transformation rules. Here we provide the definition of transformation rule as followings: 

Definition 3.2: A transformation rule r = P → (T_{pre}, T_{post}) where P defines the pattern to select the element of source model and the template pair (T_{pre}, T_{post}) defines the mapping to target model. Respectively T_{pre} defines the mapping to target model before traversing children of selected element, and T_{post} defines the mapping to target model after traversing children of selected element. The rationale of this design is closely related to the transformation algorithm that we will talk about in the next sub section.

In our framework, the development of transformation is in a large part the process of constructing transformation rules. The rule set in the Figure 2 is an extensible component. Different set of templates can be used in different transformation tasks for various target platforms. That’s why we also call the template “cartridge” to reflect the exchangeability of templates.

3.4 Transformation Algorithm

Metamodel based transformation uses the elements of metamodel. Our adopting of Hierarchical Relational Metamodel(HRM) allows us to build an internal representation of ZOOM models in a structure similar to Abstract Syntax Tree(AST). Once metamodel is generated as an AST like structure, it is accessible by the transformation process through traversing the tree.

```java
1  transformNode(node, ruleSet, outputModel)
2    rule <− findMatchingrule(node, ruleSet) //finding the matching rule for this node
3    targetText <− instantiate(rule.pre, node)
4    outputModel.append(targetText) //getting the output text by applying pre part of the rule
5    foreach c is a child of node
6      transformNode(c, ruleSet, outputModel)
7    endforeach //traverse all the children nodes and do transformation on each of them
8    targetText <− instantiate(rule.post, node)
9    outputModel.append(targetText) //getting the output text by applying post part of the rule
```

We use an algorithm of “pre-order” to traverse of the tree which means each node is visited before its children are visited and the root is visited first. The algorithm is exemplified by the pseudo code shown in the above pseudo-code.

As we can see in Definition 3.2, a transformation rule has two mapping part, T_{pre} and T_{post}. They are represented as rule.pre and rule.post in the pseudo code. Shown in the above pseudo code: rule.pre is the mapping before traversing children of selected element, while rule.post is the mapping after traversing children of selected element. And as shown in line 5-7 in the pseudo code, each node in the metamodel will be

![Class Diagram](image-url)
visited once and all its children node will get visited. To trigger the transformation algorithm, the root node of source metamodel need to be passed, in the form of $\text{transformNode}(\text{root}, \text{ruleSet}, \text{outputModel})$

### 3.5 Model Transformation Process

To start the MDE process we need to build a platform-independent model that comprises the essence of target software system. This is the only model that the developer will create completely “by hand.” The other models are mostly generated. The PIM for Community Sports League is depicted in Listing ???. It will be used as input in the following code generation examples as Roster.zoom.

The complete transformation process is depicted in Figure 4. It is divided into 5 individual steps. Now let’s walk through the transformation process step by step.

#### 3.5.1 Parsing the source model

A source model is provided, in our case, student.zoom in Listing ???. Parsing involves reading actual source code of the model, or the textual representation of model roster and splitting it into understandable language symbols. This is made possible by ZOOM’s formally defined syntax. A parser will parse the textual representation of model roster and generates an internal abstract syntax tree (AST) representation of roster, which is an object tree.

#### 3.5.2 Traversing the object tree

Once metamodel is generated as AST, which is accessible by the transformation process through traversing the tree. We use an algorithm of “pre-order” traversal of the tree which is introduced in section 4.3. It means each node is visited before its children are visited and the root is visited first. The traversing is exemplified by pseudo code shown in section 3.4. Since the matching metamodel node with rule(step 3) and generating target text(step 4) happen during the process of traversing. The pseudo code in section 3.4 actually shows all of these 3 steps. To trigger this transformation algorithm, the root node of source metamodel need to be passed. In our case, it is passing as $\text{transformNode}(\text{ml}, \text{ruleSet}, \text{outputModel})$, since $\text{ml}$ is the root node.

#### 3.5.3 Matching node in object tree with rule

The key step in this transformation process is applying rules to source models represented by their metamodels. At this point, transformation engine enters into the picture. As implied in transformation rule definition, $r = (P \rightarrow (T_{\text{pre}}, T_{\text{post}}))$, to apply a rule on a certain model include both matching the pattern $P$ and implement mapping $T_{\text{pre}}$ and $T_{\text{post}}$. The pattern $P$ is specified to make sure that right node is being located and used. In the pseudo code in section 3.4, $\text{rule.pre}$(line 3) and $\text{rule.post}$(line 8) function as a template for the transformation. When implement a template,
3.5.4 Generating target text

The generation of target text can be as simple as output the text included in the rule pre or emphpost elements as shown in pseudo code in section 3.4 line 3 and line 8. However, more complicated scenario can be involved in this step. For example, in most of the cases, expressions are in different forms in source and target model. We provide an extensible mechanism to assist the transformation, or mapping. The result of these steps is Literate Target Code. It will be used as input in the final step, post processing.

3.5.5 Post processing

The Literate Target Code generated in Step 4 as shown in Figure 4 may or may not in a desirable order that fits to the target technical platform. Post processing is responsible to rearrange the Literate Target Code in a desirable style that fits to the target technical platform. Here we treated the Literate Target Code as pieces of segment that can be flexibly rearranged. A post process goes through all these pieces and place each of them in right places in final models or code. This frees the model transformation engine from the details of contextual requirements of target platform. This approach has a similar style as proposed in Knuth’s Literate Programming [14].

After all the above transformation steps, the results are models or source code of target platform. In our example, the results are a group of Java source code.

4 Related Work

A number of partial solutions to describe and implement model transformation are currently available. Some of these are applicable only in a limited domain, or provide very low-level abstractions for transformations [15].

4.1 AndroMDA

AndroMDA [16] is a code generation tool that takes a UML model as input and generates source code as output. It adopts a template-based transformation methodology similar to ours in a degree but differs significantly in handling of metamodel. Using a series of template files (which you can customize if you wish), AndroMDA can produce source code from a UML model in any programming language. Default templates exist to generate Java code (and in particular J2EE code). AndroMDA was designed to get the information necessary to generate code from MOF compliant models inside a MOF repository.

Both AndroMDA and our approach are template-based, metamodel-based model transformation frameworks that support code generation. Both have an extensible architecture consists of cartridges. These cartridges generate the code specific to a certain concrete technical platform. However, the fundamental difference between these two approaches are the metamodel that they base upon. AndroMDA uses MOF while we use HRM. Because of the complexity of the MOF compliant metamodel, when AndroMDA traverses its AST objects, it has to access them via a proprietary JMI interfaces, metamodel facades. Comparing to AndroMDA, our approach simplified the transformation template development by adopting a concise, tree-structure metamodel.

4.2 QVT(Queries/Views/Transformations)

In the Model-Driven Architecture(MDA), QVT (Queries/Views/Transformations) is a standard for model transformation defined by the Object Management Group. Presently there are several products (commercial or open source) that claim compliance to the QVT standard. QVT defines a standard way to transform source models into target models. Duddy et al propose a transformation language which will meet the requirements of QVT RFP, and several others besides [17]. The language is declarative and patterns based. Transformation descriptions are explicitly reusable and modular. Rules that make up such descriptions may be aspect-driven, allowing for transformations to be written to address semantic concepts rather than structural features.

Since the abstract syntax of QVT conform to MOF 2.0 metamodel, one of the strength in our approach again is adopting a concise, tree-structure metamodel, HRM. Because of the simplified metamodel, when transformation engine traverses its AST objects, it can have direct access to the properties of the objects. This facilitates model transformer to develop transformation template in a easier and quicker way.
5 Conclusion

In this paper, we present a template-based model transformation framework using Hierarchical Relational Meta-model (HRM). By adopting Z-based Object-Oriented Modeling notation (ZOOM) as the formal modeling notation, this model transformation framework provides benefits consisting of: 1) readability and rigor of meta-model definitions; 2) simplicity of transformation definition; and 3) extensibility of transformation templates.

The current development of this project has made substantial progress and further research effort will be mainly focusing on two things: one is to extend the capacity of the current framework; the other is to further prove the validity of this research by building more sophisticated cases. With these two major parts in place, we can further compare our approach with other model transformation mechanisms to verify the advantages of our framework, which is providing a simple, effective, and practical way to define model transformations.

References