Operation Composition in Model Transformations with Complex Source Patterns

Arda Goknil¹, N. Yasemin Topaloglu², Klaas van den Berg³

Software Engineering Group, University of Twente, P.O. Box 217, 7500 AE, Enschede, The Netherlands
¹a.goknil@ewi.utwente.nl, ³k.g.vandenberg@ewi.utwente.nl

Department of Computer Engineering, Ege University, Izmir, Turkey
²yasemin.topaloglu@ege.edu.tr

Abstract. In model transformations, rules handle mapping between source metamodel elements and target metamodel elements. There are cases in which more complex structures in the source have to be selected. In some transformation languages, these complex source patterns require decomposition of transformation specifications into multiple rules. This decomposition negatively affects qualities of transformation specifications such as modifiability and consistency. The handling of a complex source pattern by a single transformation rule requires composition at finer granularity, i.e., operation composition. In this paper, we show how operation composition improves the quality of transformation specifications for complex source patterns. As an illustration, we implemented a complex source pattern transformation for agent systems in two well-known model transformation languages, ATL and Tefkat. ATL supports composition at rule level, and Tefkat supports composition at operation level. We describe the requirements for transformation languages in order to support the handling of complex source patterns in single transformation rules. We discuss some transformation mechanisms that provide operation composition.

Keywords: Model Transformations, Complex Source Patterns, Operation Composition, ATL, Tefkat.

1 Introduction

Model transformations have been studied to support the context of Model-Driven Engineering (MDE). Different techniques are defined for the composition of transformation rules. In some cases, it is possible to compose several transformation rules into one transformation rule. In other cases, the composition is achieved by linking several model transformations expressed in different languages and executed by different tools.

In most of these techniques, the granularity of composition is at the level of the transformation rules themselves. The transformation rule is considered as the atomic element for the transformation. Rules – whether written in different languages or in a single language – are composed in order to constitute the complete model transforma-
tion. Typically, these transformation rules use mapping between source metamodel elements and target metamodel elements [21]. For instance, model weaving [4] [5] [16] is an operation performed on source and target metamodels. Metamodels are woven by establishing typed links between their elements in order to generate model transformation code automatically. In this approach, generated rules are usually based on one-to-one and one-to-many mappings between metamodel elements. Therefore, the query part of the rule usually selects a single instance of a type from the source metamodel. In most cases, rule composition techniques focus on these kinds of rules which are identified by using this approach.

However, there are cases in which more complex structures for the source model need to be selected. For example, instances of more than one metamodel type need to be selected and filtered on the basis of the concrete property values found in the source model. There may be different interpretations of the term “complex” for such model queries. From our point of view, an informal definition of a complex pattern is that it has multiple source pattern elements, a facility to select collections (more than one model element is matched by the pattern element, e.g., all the attributes of a class are extracted in a single structure), and it has filtering (constraints) based on property values of the model being transformed. This will be explained in Section 3. Our experience [11] [12] [13] shows that complex source patterns require decomposition of transformation specifications into multiple rules and additional structures in transformation languages that do not support operation composition. This reduces two qualities of the transformation definitions: the modifiability and consistency of transformation definition. A finer grained transformation composition supported by a transformation language does not reduce these qualities.

We consider transformation operations such as create, read, update and delete operations (instead of transformation rules) as atomic parts of model transformations. The composition of these operations constitutes a distinct transformation rule for a complex source pattern. The composition granularity at the operation level reduces the number of transformation rules. This improves the qualities discussed above. This will be explained in Section 4.

We discuss the operation composition supported by transformation language mechanisms for complex source pattern transformations. As a case study, we implemented a complex source pattern transformation for a semantic web-enabled agent system [6]. We used two well-known model transformation languages: ATL[9] [9] and Tefkat [23]. They follow different transformation approaches: hybrid (ATL) and declarative logic-based (Tefkat). We chose these two transformation languages because they support composition at two different levels of granularity (rule level and operation level). In ATL, which supports only rule composition, we have to decompose our transformation specification into multiple rules and additional structures in order to cope with complex source patterns. As a consequence, changes in the source pattern are scattered over several rules. This scattering negatively affects the modifiability and consistency of transformation definitions. In order to avoid the deficiencies of the current rule composition techniques, we consider a complex source pattern as a distinct construct. We map this construct to a single module provided by Tefkat.

1 Since there are two versions of the ATL compiler currently in use (ATL 2004 and ATL 2006), ATL in this paper refers to version 2004 unless we state ATL 2006.
In this way, the mapping avoids the scattering of changes over several rules. Transformation languages should support such a finer grained composition in order to provide such mapping. Although some transformation languages such as Tefkat provide such mapping, the requirements and mechanisms to achieve operation composition are not well-studied.

In this paper, we show how a finer granularity for composition supports mapping a complex pattern transformation to a single transformation rule and how this improves solutions for complex source pattern specifications. We evaluate the implementation of a sample complex source pattern (transformation for semantic web-enabled agent system development) based on the compositions at two different levels of granularity (rule level and operation level). We explain which requirements transformation languages should fulfil in order to support the mapping of a complex source pattern transformation specification to a single transformation rule and why operation composition is needed for this support. We describe possible model transformation language mechanisms that provide operation composition and problems of implementing these mechanisms.

The paper is organized as follows. Section 2 gives a conceptual framework and some quality properties of model transformations. Section 3 presents the example in order to explain both simple and complex source patterns, and their transformations. Section 4 gives the implementations of the example in ATL and Tefkat to depict the current composition support at two different levels of granularity for complex source pattern transformations. In Section 5, we explain why operation composition is crucial and which requirements transformation languages should satisfy in order to support operation composition. Section 6 discusses possible operation composition mechanisms and proposes future work. Section 7 describes related work about existing transformation language support for operation composition. Section 8 presents our conclusions.

2 Transformation Specifications

In this section, we describe a conceptual framework and some qualities of model transformations. In Section 2.1 we describe concept diagrams for transformation specifications with a single operation, and with multiple operations and operation composition. The qualities of transformation specifications, modifiability and consistency, are discussed in Section 2.2.

2.1 Conceptual Framework for Transformation Specifications

In this subsection, we explain terms such as transformation specification, transformation rule, rule composition, operation and operation composition in terms of the transformation pattern. We also give an overview of the approach for operation composition. Figure 1 shows the transformation pattern.
Kurtev [18] defines model transformation according to the transformation pattern in Figure 1: "a model transformation is a process of automatic generation of a target model from a source model, according to a transformation specification, which is expressed in a model transformation language". Transformation execution as shown in Figure 1 takes a transformation specification written in a transformation language and a source model conforming to its source metamodel as input and transforms the source model, according to the transformation specification, into the target model. A transformation specification is a set of rules that together describe how a model in the source language can be transformed into a model in the target language [15]. For this set of transformation rules, rules are composed to constitute the transformation specification. Most transformation approaches focus on the composition of rules written in the same language or in different languages. We can denote this composition as follows:

Transformation Specification = (R1 + R2 + R3 + R4 + ... + Rn)

Transformation Specification denotes the transformation itself; R1, R2,...Rn denote transformation rules. Although we describe rule composition operators between rules with the same operator (+) in the transformation specification, there may be different composition techniques which are denoted by this rule composition operator. A transformation rule has three parts: source pattern, target pattern and action.

Transformation Rule = {Source Pattern, Target Pattern, Action}

Instances of the source pattern are transformed into instances of the target pattern according to the action. Therefore, a pattern consists of the element(s), specified by metamodel entities, in a model to be matched. Source and target patterns may be simple or complex. A complex pattern has multiple elements, a possibility to select collections (more than one model element is matched by the pattern element, e.g., all the attributes of a class are extracted in a single structure), and filtering (constraints) based on property values of the model being transformed. A simple pattern is one that does not have at least one of these features. We will discuss these features of complex patterns in more detail in the following section.

The action part of the rule is an atomic operation in most of the model transformation languages; it includes only one operation for matches of the source pattern. For
instance, the action part creates a number of target elements in the target model for the matched elements of the source pattern in the source model. When we need to define different types of operations or multiple operations of the same type for complex source patterns, we have to decompose the transformation of this pattern into multiple transformation rules in the transformation specification. Figure 2 gives a concept diagram of the transformation specification.

**Figure 2. Concept Diagram of Transformation Specifications with single Operation**

In Figure 2, the transformation specification may have multiple transformation rules. The composition link between the Transformation Specification entity and Transformation Rule entity in Figure 2 requires rule composition mechanisms provided by model transformation languages. The Transformation Rule entity has the Action entity. In some of the well-known transformation languages, such as ATL, the action part is an operation in itself and is not composed of multiple operations.

We propose having multiple operations and composing them in the action part. This composition enables us to define a transformation for a complex source pattern in a single rule. The action part of the transformation rule may include operations such as `create`, `read`, `update` and `delete` over target models. Different operations may be composed in the action part of the transformation rule as follows:

\[
Action = (Op_1 + Op_2 + Op_3 + Op_4 + \ldots + Op_n)
\]

\(Op_1, Op_2, \ldots, Op_n\) denote transformation operations; composition of these operations constitutes the action part of a single transformation rule. We describe the composition operators between operations \((Op_1, Op_2, Op_3, Op_4, \ldots, Op_n)\) with the same operator (+) in the action part. As with rule composition techniques, there may be different operation composition techniques which are denoted by this operation composition operator. A transformation operation has three parts: source pattern elements, target pattern elements and type of operation.

\[\text{Operation} = \{\text{Source Pattern Elements, Target Pattern Elements, Operation Type}\}\]

Source pattern and target pattern elements are elements of the source and target patterns defined in the transformation rule. An operation creates, reads, updates or deletes instances of the target pattern element for the match of the source pattern.
The types of composition operations (create, read, update, delete) are specified in the operation type part. Figure 3 gives an extended version of the concept diagram for the transformation specification with operation composition.

Figure 3. Concept Diagram of Transformation Specifications with Operation Composition

The extended concept diagram with the Operation entity in Figure 3 denotes compositions at two different levels of granularity: rule composition (the composition relation between the Transformation Specification entity and the Transformation Rule entity), operation composition (the composition relation between the Action entity and the Operation entity). The Operation entity and the composition relation between the Action and Operation entities in Figure 3 are only defined in model transformation languages which support multiple transformation operations in a single rule. The Operation entity is related to the Element entity which denotes source and target pattern elements; each operation is executed on specific source and target pattern elements. The composition link between the Action entity and Operation entity shown in Figure 3 requires composition mechanisms we termed Operation Composition which should be provided by model transformation languages. In Section 4, we give instances of the concept diagrams given in Figure 2 and Figure 3 for an example implemented in ATL and Tefkat.

Different techniques can be applied to implement the operation composition. Since the composition operator for operations in the action part is very similar to the composition operator for rules in the transformation specification, rule composition techniques can also be applied to operation composition. For instance, implicit rule calls [45] can be applied to operation composition as implicit operation calls. Assume we have two “create” operations in the action part of a transformation rule name Op1 and Op2, as shown below:

\[ Op1 = \{SPE1, TPE1, Type: Create\} \]
\[ Op2 = \{SPE2, TPE2, Type: Create\} \]
$SPE1$ and $SPE2$ are source pattern elements of the source pattern; $TPE1$ and $TPE2$ are the target pattern elements of the target pattern. These two operations are composed in the action part of a single transformation rule as shown below:

\[ \text{Action} = (Op1 + Op2) \]

Operations $Op1$ and $Op2$ are the operands while (+) is the composition operator of the composition in the action part. One possible technique for composing these two operations is to call the operations $Op1$ and $Op2$ implicitly according to the relations between $SPE1$ and $SPE2$.

The action part of the transformation rule should provide mechanisms for the composition of different operations in a single rule in order to increase some transformation qualities in transformations for complex source patterns. In the rest of the paper, we investigate how current rule composition techniques reduce some qualities in transformation specifications for complex source patterns and how the operation composition improves these qualities. We also identify possible problems for mechanisms of the operation composition in the action part of a single transformation rule.

### 2.2 Qualities of Transformation Specifications

Implementations of transformation specifications in model transformation languages mostly require decomposition of the transformation into multiple rules and additional structures. These multiple rules and additional structures are detrimental to certain qualities of the transformation definitions:

- **Modifiability of transformation code**: Modifiability is the ability of a system to be changed after it is implemented (and often deployed) [2]. When a change happens in the transformation definition, the impact on the existing modules should be determined and they should be updated accordingly. The modifiability of transformation code is tightly coupled to the effort that is needed to update the existing transformation modules for a change in pattern definitions. This effort can be measured as the number of transformation modules to be changed for a change in a pattern definition. The transformation decomposition may reduce the modifiability since multiple modules may be affected by a single change.

- **Consistency of transformation code**: Consistency is the degree to which different parts of the transformation do not contradict one another. When a single concept is mapped to more than one module the modules should be consistent with each other. Since each module that implements the same constraints of the same module should be checked with other modules, the number of modules affects the effort for checking the consistency of transformation code. For instance, if we have three transformation rules (rule1, rule 2, rule3) for a single concept (complex source pattern), we have to perform the checking operation three times (in other words, we have to check the consistency between rule1 and rule2 & rule1 and rule2 & rule2 and rule3). Consistency of transformation code is also related to the modifiability of transformation code because an effort is needed to check the consistency of updated transformation rules for a change in the source pattern. We show concrete examples of this consistency problem in the following sections.
3 Simple and Complex Source Patterns in Agent Systems

In this section, we give an example in order to explain simple source patterns, complex source patterns, and their transformations. Section 3.1 describes the example derived from Agent modelling. Section 3.2 gives the features of complex and simple patterns and explains the key concepts of transformations for complex and simple patterns using the example.

3.1 Transformations for Agent Systems

We present an example that requires complex source patterns for transforming models of Agent systems. We transform platform-independent Agent models to platform-specific Agent models based on the semantic web. The target platform is the semantic web-enabled agent framework called SEAGENT [6]. The source metamodel is the Agent metamodel presented in [11]. It defines common entities that a multi agent system should have. The SEAGENT platform is implemented in Java. Therefore, the output of the transformation is a Java model based on a Java metamodel. In this paper, we focus on the interaction between agents and semantic web services in agent modelling. Figure 4 gives part of the source metamodel that captures this interaction. It extends FIPA Modelling TC’s Agent Class Superstructure Metamodel (ACSM) [24] and UML Semantic Web Service Profile (USWSP) [8].

![Figure 4. Metamodel for Agent and Semantic Web Service Interaction](image_url)
The main elements of this part of the metamodel are:

- **Agent** – the class of all agents that populate a system. It is an extension of the `Agent` class in ACSM.
- **Role** – denotes roles the agents are capable of playing at a given time.
- **Plan** – represents reusable task structures for agents. Plans may be associated with `Interface`, `Process`, and `Grounding` elements to access the service definitions.
- **Semantic Web Service** – represents any service whose capabilities and interactions are semantically described within a Semantic Web-enabled multi-agent system.

Java classes and objects are concrete realizations of these entities at the platform-specific level. Figure 5 presents an example of a Java metamodel for the Java language.

---

**Figure 5. Metamodel for Java**

In order to save space we consider only one transformation. We transform agents that invoke semantic web services via their plans. The source pattern has three main pattern elements: **Semantic Web Agent**, **Semantic Service Executor Plan**, and **Semantic Service Matchmaker Agent**. Instances of these elements are transformed to `JClass` instances. Below we give the source pattern and its constraints in a language-independent notation.

```java
1  agnt1: Agent {agnt1.type = "Semantic Web Agent"}
2  role1: Role {role1.type = "Domain Role"}
3  agnt2: Agent {agnt2.type = "Semantic Service Matchmaker Agent"}
4  role2: Role {role2.type = "Registry Role"}
5  pln: Plan {pln.type = "Semantic Service Executor Plan"}
6  CONSTRAINTS {
7     agnt1.size() = 1 AND role1.size() = 1 AND agnt2.size() = 1
8     AND role2.size() = 1 AND pln.size() = > 1 AND agnt1.plays = role1
}
An Agent (agnt1) must have at least one Semantic Service Executor Plan (agnt1.applies = pln) and exactly one Domain Role (agnt1.plays = role1) to be transformed as a Semantic Web Agent. An Agent (agnt2) must fulfill some constraints to be transformed as a Semantic Service Matchmaker Agent. The Semantic Service Matchmaker Agent must have at least one Semantic Service Executor Plan (agnt2.applies = pln) and exactly one Registry Role (agnt2.plays = role2). A Plan must be related to a Semantic Web Agent and a Semantic Service Matchmaker Agent to be transformed as a Semantic Service Executor Plan. There may be more than one Semantic Service Executor Plan that fulfills these constraints in one match of the pattern (pln.size() = > 1). If the number of instances is more than one, then we want to include all of them as a single collection in the match of the pattern. Please note that the elements that are matched are identified not only on the basis of their metaclass but also with a certain value of the type attribute. Below we give the target pattern for the source pattern and the assignments between target and source pattern elements in a language independent notation.

```
1  agnt1J: JClass {agnt1J.size() = 1}
2  agnt2J: JClass {agnt2J.size() = 1}
3  plnJ: JClass {plnJ.size() = > 1}
4  ASSIGNMENTS {
5    agnt1J.name = Join("_", agnt1.name, agnt1.type);
6    agnt2J.name = Join("_", agnt2.name, agnt2.type);
7    plnJ.name = Join("_", pln.name, pln.type);
8    agnt1J.associatedClass = plnJ;
9    agnt2J.associatedClass = plnJ;
10 }
```

According to the target pattern, the transformation created one JClass instance (agnt1J) for one Semantic Web Agent (agnt1), one JClass instance (agnt2J) for one Semantic Service Matchmaker Agent (agnt2) and one JClass instance (plnJ) for one Semantic Service Executor Plan (pln). Since there may be more than one Plan in a single match of the source pattern, there may be more than one JClass instance for these Plans (plnJ.size() = > 1). JClass instances created for the Plans in the source model should be associated for JClass instances created for Agents in the source model (agnt1J.associatedClass = plnJ, agnt2J.associatedClass = plnJ). In order to save space, in Figure 4 (above), we do not consider Process, Interface and Grounding meta classes in the source and target patterns. Figure 6 depicts two instance models for the source pattern.
3.2 Simple and Complex Source Patterns in Transformations

In this section, we give the features of complex and simple patterns in transformations, using the example described in Section 3.1.

We perceive the patterns described in Section 3.1 as complex source patterns. We define the following four features in order to classify a pattern as complex:

- It has multiple source elements (in the example, they are denoted by $agnt1$, $agnt2$, $role1$, $role2$, $pln$)
- Eventual selection of collections (corresponds to $pln$ in our example)
- Global filters (constraints): Boolean conditions that involve properties of more than one source element
- Selection on the basis of property values from the model being transformed (in the example, $type$ attribute values)

Unlike complex source patterns, simple pattern definitions do not consider the dependencies between pattern elements. For instance, we can define a transformation that transforms every Agent instance to a JClass without checking the relations with other model elements like Plan instances. Without these dependency constraints, simple patterns do not have to consider global filters, selection on the basis of property values from the model being transformed, multiple pattern elements and selection of collections together in the same pattern definition.
Our definition of complex patterns in model transformations is based on the distinction between metamodel and model level mappings. Transformation rules generated from metamodel level mappings are usually based on one-to-one and one-to-many mappings between metamodel elements; those generated from model level mappings are usually based on mappings between model elements with the features given above instead of metamodel element mappings. Based on this distinction, these four features of a pattern are necessary and sufficient. A source pattern should satisfy all these features in order to be classified as a complex source pattern. For instance, a pattern that has multiple source elements, global filters, selection on the basis of property values but does not have selection of collections is not a complex pattern. Since there may be different interpretations of the term complex for pattern definitions in model transformation, we do not claim that these features are necessary and sufficient for a more general complex pattern definition from a broader perspective.

In most cases, current composition techniques provide proper support for composition of rules with simple source patterns. In order to identify transformation rules based on simple source patterns, we can use mapping between source and target metamodel elements. Model weaving [4] [5] [16] is an operation on source and target metamodels. Metamodels are woven by establishing typed links between their elements in order to generate model transformation code automatically. In this approach, generated rules are usually based on one-to-one and one-to-many mappings between metamodel elements. Therefore, the query part of the rule usually selects a single instance of a type from the source metamodel. In [21], the generation of transformation definitions from a metamodel mapping is formalized as shown below: Given $M_1(s)/M_a$, $M_2(s)/M_b$, and $C_{M_a} \Rightarrow M_b/M_c$, where $M_1$ is a model of a system $s$ created using the metamodel $M_a$, $M_2$ is a model of the same system $s$ created using the metamodel $M_b$, and $C_{M_a} \Rightarrow M_b$ is the mapping between $M_a$ and $M_b$ created using the metamodel $M_c$, then a transformation specification can be defined as the function $\text{Transformation}(M_1(s)/M_a, C_{M_a} \Rightarrow M_b/M_c) \Rightarrow M_2(s)/M_b$. Figure 7 shows the mapping mechanism between source and target metamodel elements.

**Metamodel Mapping**

![Figure 7. Mapping between Source and Target Metamodels](image)

$S1, S2, S3, T1, T2$ and $T3$ are source and target metamodel elements in Figure 7; $SRL1, SRL2, TRL1$ and $TRL2$ are links between these elements. Binary relations
named $M_1$, $M_2$ and $M_3$ map source metamodel elements to the corresponding target metamodel elements. These binary relations are: $M_1 = (S_1, T_1)$, $M_2 = (S_2, T_2)$, $M_3 = (S_3, T_3)$. The mapping between source and target metamodel is the composition of these binary relations: \( \text{Mapping} = (M_1 + M_2 + M_3) \). If we map the Agent metamodel in Figure 4 to the Java metamodel in Figure 5, we get these binary relations between metaclasses: $M_1 = (\text{Agent}, \text{JClass})$, $M_2 = (\text{Plan}, \text{JClass})$, $M_3 = (\text{Interface}, \text{JClass})$, $M_4 = (\text{Process}, \text{JClass})$, $M_5 = (\text{Grounding}, \text{JClass})$. Figure 8 shows these mappings between some parts of the Agent and Java metamodels in Figure 4 and Figure 5.

Transformation rules can be generated from these mappings given in Figure 8. Lopes et al. [21] separate the mapping specification and transformation specification. However, this distinction can be considered as the action part of the transformation rule in a transformation specification. In this manner, a transformation rule specifies only mappings between source and target pattern elements without the action part. These are the transformation rules generated from the metamodel mapping of Agent and Java metamodels in Figure 8 by adding the action parts to the mapping relations:

- $R_1 = \{\text{Agent, JClass, Action: Create}\}$
- $R_2 = \{\text{Plan, JClass, Action: Create}\}$
- $R_3 = \{\text{Interface, JClass, Action: Create}\}$
- $R_4 = \{\text{Process, JClass, Action: Create}\}$
- $R_5 = \{\text{Grounding, JClass, Action: Create}\}$

Each transformation rule above has a source pattern, a target pattern and an action. Since the source patterns generated from the metamodel mapping are simple source patterns, each source pattern in rule definitions has only one source pattern element.
typed as metamodel entities (Agent, Plan, Interface, Process, Grounding). They do not consider the features of complex source patterns together in the same pattern. The transformation specification is the composition of these transformation rules:

\[ \text{Transformation Specification} = (R1 + R2 + R3 + R4 + R5). \]

The implicit rule calls defined in [1] may solve the problems of the composition and order of execution of mapping rules in the execution of transformation rules. For instance, the composition operator in a Transformation Specification between R1, R2, R3, R4 and R5 is supported by these implicit calls in ATL [9] during the execution of transformation rules.

Figure 9 depicts the execution, independent of the transformation engine, of the specification derived from the metamodel mapping given in Figure 8.

![Figure 9. Transformation Execution based on Metamodel Mapping](image)

The source model (the Agent model) in Figure 9 has one instance of the Agent metaclass and two instances of the Plan metaclass. Therefore, transformation rules R1 and R2 generated from mapping of the Agent and Plan metaclasses with the JClass metaclass are executed in the transformation execution. The transformation execution is the composition of these two rules for Figure 9:

\[ \text{Transformation Execution} = (R1 + R2) \]

Depending on the execution engine, the transformation rule R2 can be executed twice (one for the PriceBargainerPlan instance, one for the ProductBuyerPlan instance), or it can be executed once (one execution covers two Plan instances with selection of collections). If the transformation rule R2 is executed twice, the transformation execution is:

\[ \text{Transformation Execution} = (R1 + R2 + R2) \]

In both cases, transformation languages should support mechanisms in rule composition granularity to compose these rules in transformation execution. For instance,
in an implicit operation call, \( \text{R1} \) is called by \( \text{R2} \) or \( \text{R2} \) is called by \( \text{R1} \) implicitly by using the relation between the \textit{Agent} and \textit{Plan} instances in Figure 9.

There are cases, however, in which more complex structures in the source model should be selected. Instances of more than one metamodel type need to be selected and filtered on the basis of the concrete property values found in the source model. For instance, the transformation for the Agent-Semantic Web Service interaction pattern given in Section 3.1 has different features (such as multiple source elements and global constraints) than metamodel mapping transformations. Rule granularity for transformation composition that is offered by metamodel mapping may not provide an efficient support to define complete complex source pattern transformations. Solutions based on this rule composition granularity have certain reduced qualities of the transformation specification like modifiability of transformation code, and consistency of transformation code.

We need a finer grained mapping to reduce the complexity of transformations. This requires a more low-level composition granularity for model transformations generated from model level mapping. We consider transformation operations such as \textit{create}, \textit{read}, \textit{update} and \textit{delete} operations instead of transformation rules as atomic parts of model transformations for a better composition granularity. The synthesis of these operations constitutes a single transformation rule. This synthesis of operations in a single rule requires additional composition and integration mechanisms in model transformation languages. With these mechanisms, it is possible to transform one complex source pattern into a target pattern in a single rule.

The generation of complex pattern transformations is slightly different from the generation of simple pattern definitions from metamodel mapping. We formalize this generation as shown below:

Given \( \text{IP}_1(\text{M}_1)/\text{P}_1 \), \( \text{IP}_2(\text{M}_2)/\text{P}_2 \) and \( \text{CP}_{1 \rightarrow 2}/\text{P}_3 \), where \( \text{IP}_1 \) is an instance pattern in \( \text{M}_1 \) created using the pattern \( \text{P}_1 \) as a query in \( \text{M}_1 \), \( \text{IP}_2 \) is an instance pattern in \( \text{M}_2 \) created using the pattern \( \text{P}_2 \) as a query in \( \text{M}_2 \), and \( \text{CP}_{1 \rightarrow 2} \) is the mapping between \( \text{P}_1 \) and \( \text{P}_2 \) patterns created using the mapping pattern \( \text{P}_3 \). A transformation rule for these source and target patterns can be defined as the function \( \text{TransformationRule}(\text{IP}_1(\text{M}_1)/\text{P}_1, \text{CP}_{1 \rightarrow 2}/\text{P}_3) \rightarrow \text{IP}_2(\text{M}_2)/\text{P}_2. \) The function \( \text{TransformationRule} \) denotes a transformation rule that transforms a source pattern into a target pattern itself. Therefore, the complete transformation is the execution of the set of these transformation rules formalized by the function \( \text{TransformationRule} \). One difference in this mechanism is that the basis for the transformation is the source and target pattern mappings in the model level in a single rule instead of metamodel mapping. The mappings between \( \text{IP}_1 \) and \( \text{IP}_2 \) patterns can be implemented as transformation operations and the composition of these operations constitutes the transformation rule which transforms the complex pattern definition. In this structure, a transformation rule should support all entities (especially the \textit{Operation} entity and the composition relation between the \textit{Action} and \textit{Operation} entities) in the concept diagram of the transformation specification in Figure 3 for the conceptual transformation framework.
4 Implementations of Agent System Transformations

In this section, we implement the example described in Section 3.1 in two well-known model transformation languages\(^2\) (ATL [9], and Tefkat [23]) to explain the transformation specifications for complex source patterns based on current rule composition techniques given in Section 3.2. The choice is motivated by the fact that they represent different transformation approaches for composition mechanisms thus increasing the scope of our findings. ATL allows both declarative and imperative transformation styles. Tefkat adopts a declarative paradigm. They support composition granularity at two different levels (rule and operation levels). We do not discuss the language features in details.

Implementations of complex source pattern transformations in transformation languages such as ATL, relying only on current rule composition techniques for simple patterns, require decomposition of the transformation into multiple rules and additional structures. These multiple rules and additional structures for the transformation of the same pattern reduce certain qualities, as discussed in Section 2.2 (modifiability and consistency of transformation code). In order to avoid the deficiencies of the current rule composition techniques, we consider a complex source pattern as a distinct construct. We map this construct to a single module provided by Tefkat.

Section 4.1 gives the example implementation in ATL and explains the reduced qualities for the implementation. Section 4.2 gives the sample implementation in Tefkat and explains how the operation composition in Tefkat improves these two qualities. In Section 4.3, we present a change scenario for the sample implementations in order to show how two different composition granularities affect these two qualities.

4.1 Implementation in ATL

There are two versions of the ATL compiler currently in use: ATL 2004 and ATL 2006. ATL 2004 allows only one pattern element in the query part of rules. Therefore, we can only define one-to-one and one-to-many transformations in a single rule. ATL 2006 allows multiple source elements and has expressivity closer to that provided by Tefkat. We intentionally use ATL 2004 to illustrate the composition problems caused by simple patterns.

The transformation rule in ATL has a \textit{from} section for the source pattern part which allows only one pattern element and a \textit{to} section for the target pattern part which allows multiple pattern elements. The transformation rule does not have an explicit action part and the action part allows only one operation. In fact, the action part of the rule is an operation itself. Since ATL does not have a direct support for in-place model updates, it only considers \textit{create} operations.

To implement the example with simple patterns only, we have to define rules for each element in the source pattern. In fact, we implement a more complex pattern

\(^2\) Implementations of the example in these two languages are available at: http://www.home.cs.utwente.nl/~goknila/
than that in Section 3.1 since it is closer to the original transformation explained in [12]. We define six rules to transform the complex source pattern given in Section 3.1.

The ATL rules for the example are: rule SemanticWebAgent2AgentJ, rule SemanticMatchmakerAgent2AgentJ, rule ExecutionPlan2ExecutionJPlan, rule Interface2OWLInterface, rule Process2OWLProcess, and rule Grounding2OWLGrounding. Each rule transforms only one source model element to one or more target model elements. We only show two rules:

```
1 rule SemanticWebAgent2AgentJ {  
2  from ag: Agent!Agent(  
3    ag.partofPatternforWebAgent )  
4  to c:JAVA!Class (  
5    name<- ag.type,  
6    associatedClass<-ag.executorPlans )  
7 }  

1 rule SemanticMatchmakerAgent2AgentJ {  
2  from ag: Agent!Agent (  
3    ag.partofPatternforMatchmaker )  
4  to c:JAVA!Class (  
5    name<- ag.type,  
6    associatedClass<-Sequence{ag.semanticInterfaces,  
7      ag.executorPlans} )  
8 }
```

In order to capture the constraint in the pattern that requires a relation to other instances like Domain Role, Semantic Service Executor Plan, Semantic Service Matchmaker Agent, and Registry Role we provide a filter implemented as the helper partofPatternforWebAgent called in line 3 of the rule SemanticWebAgent2Agent. Another filter implemented as the helper partofPatternforMatchmaker is called in line 3 of the rule SemanticMatchmakerAgent2AgentJ. Similar helper rules and constraint repetitions are required for other rules in the transformation.

```
1 helper context Agent!Agent def:  
2   partofPatternforWebAgent : Boolean =  
3     if self.type='Semantic Web Agent' and  
4       self.role.type='Domain Role' and self.plan->  
5       select(p|p.type='Semantic Service Executor Plan'  
6         and p.agentP->exists(ag|ag.type='Semantic Service  
7           Matchmaker Agent' and ag.role.type='Registry Role'))->  
8       forAll(p|p.interfaceP.semanticI = p.processP.semanticP  
9         and p.interfaceP.semanticI = p.groundingP.semanticG  
10         and p.interfaceP.agentI.plan->exists(pln|pln = p)) then  
11         true  
12     else false  
13   endif;  

1 helper context Agent!Agent def:  
2   partofPatternforMatchmaker : Boolean =  
3     if self.type = 'Semantic Service Matchmaker Agent' and
```
self.role.type='Registry Role' and self.interface->
forAll(i|i.planI.type='Semantic Service Executor Plan' and
i.planI.processP.semanticP = i.semanticI and
i.planI.groundingP.semanticG = i.semanticI and i.planI.agentP->
exists(agnt|agnt = self) and i.planI.agentP->
exists(agnt|agnt.type = 'Semantic Web Agent' and
agnt.role.type = 'Domain Role')) then
true
else false
endif;

These helpers are the realization of the constraints part of the source pattern in Section 3.1. The helper partofPatternforWebAgent is used to select the Agent instances for Semantic Web Agents in pattern matching. The condition in line 3 of the helper partofPatternforWebAgent checks the type attribute of Agent instances; the conditions between line 4 and line 10 check the relations of Agent instances with other instances like Semantic Service Executor Plan and Semantic Service Matchmaker Agent. They determine if the matched agent as Semantic Web Agent is a part of the pattern. The helper partofPatternforMatchmaker is used to select the Agent instances for Semantic Matchmaker Agents in pattern matching. The condition in line 3 of the helper partofPatternforMatchmaker checks the type attribute of Agent instances; the conditions between line 4 and line 10 check the relations of Agent instances with other instances like Semantic Service Executor Plan and Semantic Service Matchmaker Agent. They determine if the matched agent as Semantic Matchmaker Agent is a part of the pattern. These two helpers have the same conditions for the same pattern to check different pattern elements as a part of the pattern. For every rule, we have to implement a “partOf” helper that determines if the instance element is a part of the pattern. There are six helpers in this transformation to select the appropriate model instances.

Another helper is used to select the elements for creating relations between target elements. For instance, there is a relation between the Agent and Plan classes in Figure 4. In the transformation, the assignment should be performed to define the relation between the Java class for Semantic Web Agent and the Java class for Semantic Service Executor Plan. There might be other Plan instances which are not Semantic Service Executor Plan. Therefore, the appropriate Plan instances which are Semantic Service Executor Plan should be selected for association assignment. The assignment in line 6 of the rule SemanticWebAgent2AgentJ calls a helper named executorPlans to select the Semantic Service Executor Plans and create the relation between corresponding target elements. We implemented four helpers for these relations. The helper executorPlans is given below:

helper context Agent!Agent def:
executorPlans : Sequence(Agent!Plan) =
self.plan->select(p| p.type = 'Semantic Service Executor
Plan' and p.agentP->exists(agn|agn.type='Semantic Web
Agent' and agn.role.type='Domain Role') and p.agentP->
exists(agn|agn.type='Semantic Service Matchmaker
Agent' and agn.role.type='Registry Role' and
agn.interface->includes(p.interfaceP))) and
p.interfaceP.semanticI = p.processP.semanticP and
p.interfaceP.semanticI = p.groundingP.semanticG);

The condition in line 3 checks the type attribute of Plan instances; the conditions between line 4 and line 10 check the relations of Plan instances with other instances like Semantic Web Agent and Semantic Service Matchmaker Agent.

Figure 10 gives a concept diagram of transformation specification of the example implementation in ATL based on the concept diagram given in Figure 2 in order to show how ATL supports only rule composition.

![Transformation Specification Diagram](image)

**Figure 10. Instantiated Concept Diagram of the Implementation in ATL**

Here, we only consider two ATL rules (SemanticWebAgent2AgentJ and SemanticWebAgent2AgentJ) of the example implementation and skip other transformation rules (ExecutionPlan2ExecutionJPlan, Interface2OWLInterface, Process2OWLProcess, and Grounding2OWLGrounding). The entity TransformationSpecification corresponds to the transformation itself and has two rules (SemanticWebAgent2AgentJ and SemanticWebAgent2AgentJ). Each rule has a source and target pattern. Since ATL makes it possible to define one-to-one and one-to-many transformations, source and target patterns of these two rules do not satisfy the conditions for a complex pattern, as described in Section 3.1. Decomposing a complex pattern transformation into multiple rules is also decomposing a complex pattern definition into multiple simple pattern definitions. Therefore, the types of these two patterns are simple. Each transformation rule has an action. In Figure 10, an action corresponds to a single create operation between elements matched by source and target patterns. Since the transformation rule structure in ATL makes it possible to define only one operation that corresponds to an action in a single rule, ATL supports the composition
granularity at the rule level, which is denoted as a composition link between the
TransformationSpecification and TransformationRule entities in Figure 10.

Figure 11 gives a sketch of the rules and helpers for the implementation in ATL. It
gives the rule definitions and which helpers are called in which rule. For instance,
partOfPatternforWebAgent helper is called in the “from” section of the rule Semantic
WebAgent2AgentJ as a query for the complex pattern.

Figure 11. Rules and Helpers for the Implementation in ATL

There are two alternative types of rule invocation to compose ATL rules in the
transformation: implicit rule calls and explicit rule calls. In our case study, we used
implicit rule calls (in line 6 of the rule SemanticWebAgent2Agent, and in line 6-7 of
the rule SemanticMatchmakerAgent2Agent). This leads to more adaptable transforma-
tion definitions and to loosely coupled rules [9]. Implicit rule invocation may provide
a complete support for rule composition for simple source patterns as explained in
Section 3.2. The pattern in the example, however, has different features than simple
source patterns such as multiple source elements, the selection of collections, and
filtering based on property values of these elements being transformed. Therefore, we
need the helpers to compose rules for the same source pattern. Rules will query the
same pattern for different pattern elements with the help of these extra structures.

These extra structures for rule composition reduce certain qualities of transforma-
tion definitions: modifiability and consistency of transformation code. The decompo-
sition of the transformation to multiple rules and helpers reduces the understandabil-
ity. There is no explicit grouping of these modules that indicates their relation to the
intended source pattern. It is difficult to trace the constructs back to the single pattern.
For modifiability of transformation definitions, assume we want to change the source
pattern by modifying constraints or relations among the pattern elements. Since the
source pattern is implemented in multiple rules and helpers, most of them will be
affected. For instance, if we need to change the constraint for Plans, it is expected to
change only the helper partOfPatternforExecutorPlan written to check Plans. Since every helper defines the constraints for the entire source pattern, we have to update all helpers for this change. This makes changes prone to error and reduces modifiability. Another consequence of inability to express the complex source pattern in a single module is the reduced consistency of transformation code. In our implementation, every helper contains the constraints for relations among the pattern elements. Therefore, there is a repetition of code across the helpers. It is essential that helpers are consistent with each other regarding these relations. If a mistake is made even in a single helper (thus breaking the consistency) the query logic will be incorrect. This problem is again a consequence of the inability to express the complex source pattern in a single module.

4.2 Implementation in Tefkat

Tefkat is a declarative, logic-based transformation language. The source part of the rules allows definition of more than one pattern element and constraints on these elements. The language does support selection of collections in the source part. The ability to define more than one source pattern element and select collections reduces the number of rules in the example implementation (contrary to the ATL implementation). Therefore, this ability allows us to capture the source pattern into a single rule.

The transformation rule in Tefkat has the `FORALL` section for the source pattern part and the `MAKE` section for the target pattern part. These two sections both allow multiple pattern elements in pattern definitions. Since Tefkat supports selection of collections for the pattern elements in its source pattern part, the action part allows multiple operations for all pattern elements in the source pattern. Tefkat does not perform in-place model updates. Therefore, it only considers create operations.

We define one single rule to transform the complex source pattern given in Section 3.1. The Tefkat rule AgentServiceInteraction for the example is the following:

```
1 RULE AgentServiceInteraction
2   FORALL Agent ag1 {
3     type: "Semantic Web Agent";
4     role: Role role1 {
5       type: "Domain Role";
6     };
7   plan: Plan pln {
8     type: "Semantic Service Executor Plan";
9     interfaceP: Interface intf {
10        agentI: Agent ag2;
11        semanticI: SemanticWebService sws;
12     };
13     processP: Process prc {
14        semanticP: SemanticWebService sws;
15     };
16     groundingP: Grounding grd {
17        semanticG: SemanticWebService sws;
18     };  
19   };
```
The rule `AgentServiceInteraction` has nine source pattern elements in its source pattern part (Agent `ag1`, Agent `ag2`, Plan `pln`, Role `role1`, Role `role2`, Interface `intfc`, Process `prc`, Grounding `grd`, SemanticWebService `sws`). The element `ag1` matches all instances of Agent whose type is “Semantic Web Agent” and role is “Domain Role” (Statements between line 2 and line 6 check this condition). The Agent instance matched by the `ag1` pattern element is also related to one Semantic Service Executor Plan instance (or multiple Plan instances) matched by the `pln` source pattern element (line 7). The rule `AgentServiceInteractionRule` asserts that a JClass with the same
The number of matched elements for Semantic Service Execution Plan classes (selection of collections for Plans) is different from the number of matched classes for Semantic Web Agent and Semantic Matchmaker Agent classes (only one instance for each Agent). Since we define the \textit{pln} pattern element for Semantic Service Executor Plan instances in the definition of the \textit{ag1} pattern element (\textit{ag1} is between line 2 and line 20; \textit{pln} is between line 7 and line 19), the \textit{AgentServiceInteractionRule} matches one Semantic Web Agent instance and one or more Semantic Service Executor Plan instances in a single match. The source pattern element \textit{pln} matches all instances of Plan whose type is “Semantic Service Executor Plan” (line 8) and which is also related to Process, Interface and Grounding instances (“interfaceP: Interface intfc” in line 9, “processP: Process prc” in line 13, “groundingP: Grounding grd” in line 16). For every matched instance of Interface, Process and Grounding classes, the rule creates a JClass instance in the target model and assigns their values. The Plan instances matched by the \textit{pln} pattern element are also related to the same Semantic Service Matchmaker Agent instance matched by the \textit{ag2} source pattern element (“plan: Plan pln” in line 26). The rule \textit{AgentServiceInteractionRule} asserts that a JClass (“Class plnJ1 FROM op2(pln)” in line 32) with the same attributes of Semantic Service Executor Plan instance (“name: join("\_", pln.name, pln.type)” in line 33) and other JClasses (\textit{discoverJ} in line 39, \textit{enactJ} in line 43, \textit{engageJ} in line 47) related to Executor Plan JClass (“aggregateClass: plnJ1” assignments in lines 41, 45 and 49) must exist in the target model.

Figure 12 gives a concept diagram of the transformation specification for the example implementation in Tefkat based on the concept diagram given in Figure 3 in order to show how Tefkat supports rule composition and operation composition.
Due to limitations of space, we consider only two source pattern elements (ag1 and pln) of the rule AgentServiceInteraction and skip other source pattern elements (ag2, intf, prc, grd, role1 and role2). The entity TransformationSpecification corresponds to the transformation itself and has one transformation rule (AgentServiceInteraction). The rule has a source and target pattern whose type is complex. Since Tefkat supports multiple source elements, selection of collections, global filters and selection on the basis of property values in a single rule, we could define the transformation of this complex source pattern in a single rule. Unlike the source and target patterns in Figure 10, the source and target patterns in Figure 12 have multiple pattern elements. Here, the Create operation between pattern elements is defined as an Operation entity instead of Action entity. The Action entity in Figure 12 is used to represent the composition operator for operations defined in the AgentServiceInteraction rule. Therefore, Tefkat supports the composition granularity at the operation level which is denoted as a composition link between the Action entity and Operation entities in Figure 12.

Modifiability and rule consistency are not reduced in Tefkat. Assume we want to change the constraint for Plans (similarly to the case with the ATL implementation). We expect that this change will affect only the transformation rule AgentServiceInteractionRule written to check Plans. Since only one transformation rule defining the constraints for the entire source pattern given in Section 3.1 exists, we have to update only this rule for this change. Compared with the ATL implementation, we need less effort in implementing the change because we have less constructs to update. In Section 4.3, we give a more detailed change scenario for the implementations of the
example in ATL and Tefkat to depict the qualities of the transformation definitions in two different composition granularities.

4.3 Evaluation of Quality Properties based on a Change Scenario

In this subsection, we give a change scenario for the sample implementations in order to show how two different composition granularities affect two qualities: modifiability of transformation code and consistency of transformation rules.

When we update our complex source pattern definition given in Section 3.1 according to the change scenario we define in this section, the impact on the existing modules should be determined and they should be updated accordingly. Therefore, the modifiability of transformation code is tightly coupled to the effort that is needed to update the existing transformation modules for a change in pattern definitions. We measure this effort as the number of transformation modules to be changed for the change in the complex source pattern definition. Since each module that implements the same constraints of the same module should be checked with other modules, the number of modules affects the effort required for checking the transformation rule consistency. When a change is specified for our complex source pattern transformation, an effort is needed to check the consistency of updated transformation rules for this change in the source pattern definition. Therefore, we use a change scenario to show these affected qualities for the sample implementations at two different composition granularities.

In our change scenario, we have a change for the role types of Semantic Service Matchmaker Agents. In the source pattern definition given in Section 3.1, every Semantic Matchmaker Agent must have a Role whose type is Registry Role. In the current case, Registry Role is used for agents registering semantic web services and other agents, and matches registered semantic web services and agents according to the given parameters. In the new case, we have another source pattern named AgentRegistration. This pattern queries agents and web services registered to these agents. In the pattern AgentRegistration, Registry Role is used for agents registering web services. Therefore, we need to change the type of the role for Semantic Service Matchmaker Agents. The new role type for Semantic Service Matchmaker Agents is Registry and Matchmaker Role. The new source pattern definition is the following:

1  agnt1: Agent {agnt1.type = "Semantic Web Agent"}
2  role1: Role {role1.type = "Domain Role"}
3  agnt2: Agent {agnt2.type = "Semantic Service Matchmaker Agent"}
4  role2: Role {role2.type = "Registry and Matchmaker Role"}
5  pln: Plan {pln.type = "Semantic Service Executor Plan"}
6  CONSTRAINTS {
7   agnt1.size() = 1 AND role1.size() = 1 AND agnt2.size() = 1
8   AND role2.size() = 1 AND pln.size() = > 1 AND agnt1.plays = role1
9   AND agnt2.plays = role2 AND agnt1.applies = pln
10  AND agnt2.applies = pln
11  }
The changed part of the source pattern definition given above is the line that checks the role type of Semantic Web Service Matchmaker Agents. The role type checking part (role2.type = "Registry Role") in the source pattern definition given in Section 3.1 is changed to (role2.type = "Registry and Matchmaker Role"). When we implement this change in the transformation definition written in ATL, we have to update ten helpers (partofPatternforMatchmaker, partofPatternforWebAgent, partofPatternforExecutionPlan, partofPatternforInterfaceOWL, partofPatternforProcessOWL, partofPatternforGroundingOWL, executorPlans, semanticInterfaces, semanticWebAgent, matchmakerAgent) implementing the updated source pattern. The updated partofPatternMatchmaker helper is given below:

1 helper context Agent!Agent def:
2   partofPatternforMatchmaker : Boolean =
3   if self.type = 'Semantic Service Matchmaker Agent' and
4      self.role.type='Registry and Matchmaker Role' and
5      self.interface->forAll(i|i.planI.type=
6         'Semantic Service Executor Plan' and
7         i.planI.processP.semanticP = i.semanticI and
8         i.planI.groundingP.semanticG = i.semanticI and
9         i.planI.agentP->
10        exists(agnt|agnt = self) and
11        i.planI.agentP->
12        exists(agnt|agnt.type = 'Semantic Web Agent' and
13        agnt.role.type = 'Domain Role')) then
14       true
15   else false
16 endif;

In line 4 of the helper partofPatternMatchmaker, (self.role.type = 'Registry Role') is changed to (self.role.type = 'Registry and Matchmaker Role'). In ten helpers implementing the updated source pattern definition, we changed this line that checks the role type for Semantic Service Matchmaker Agents. It is necessary to update ten transformation units (a transformation unit is a helper in this case) for a minor change in this complex source pattern definition. This is one of the patterns of transformations for model-driven development of semantic web-enabled multi agent systems [11] [12] [13]. We analysed a subset of the system and defined many other pattern definitions such as OntologyMediation, ContentInterpretation, AgentRegistration and ServiceRegistration. In the complete transformation, we expect more than approximately 40 source pattern definitions and 300 helpers implemented in ATL. Detecting the affected helpers among 300 helpers for a small change in a source pattern definition and updating them need more effort than detecting an affected single transformation unit and updating it need. Decomposing a complex source pattern transformation into multiple rules and helpers reduces the modifiability of transformation definitions since multiple helpers are affected by a single change.

Transformation rule consistency is twofold. When we implement ten helpers, we have to be sure that these helpers implement the same source pattern definition. Therefore, we have to check the Object Constraint Language (OCL) [27] constraints for each helper to ensure they are consistent with the source pattern definition given in Section 3.1. When there is a change in the source pattern definition and we implement the change in the helpers, we also have to check the new OCL constraints for
each helper to ensure they are consistent with the new source pattern definition. Since the change is a minor one in our change scenario, we can easily check the consistency of the helpers. But when we have more complex changes, such as adding new pattern elements to the source pattern definition, we may need some automated techniques for consistency checking.

Since the complex source pattern transformation is implemented in a single rule, named `AgentServiceInteraction` in Tefkat, we have to update only this rule when we implement this change in the transformation definition written in Tefkat. The updated `AgentServiceInteraction` rule (only the source pattern definition part) is given below:

```plaintext
1 RULE AgentServiceInteraction
2   FORALL Agent ag1 {
3       type: "Semantic Web Agent";
4       role: Role role1 {
5           type: "Domain Role";
6       };
7       plan: Plan pln {
8           type: "Semantic Service Executor Plan";
9           interfaceP: Interface intfc {
10              agentI: Agent ag2;
11              semanticI: SemanticWebService sws;
12        };
13           processP: Process proc {
14              semanticP: SemanticWebService sws;
15        };
16           groundingP: Grounding grd {
17              semanticG: SemanticWebService sws;
18        };
19   },
20  Agent ag2 {
21      type: "Semantic Service Matchmaker Agent";
22      role: Role role2 {
23           type: "Registry and Matchmaker Role";
24       };
25  plan: Plan pln ;
26  }
```

In line 24 of the rule `AgentServiceInteraction`, (type: "Registry Role") is changed to (type: "Registry and Matchmaker Role"). It is necessary to update only one transformation unit (is the same as a transformation rule in this case) for a change in this complex source pattern definition. Since all these patterns can be considered as single concepts, and most of them are mapped to a single transformation rule in Tefkat, the change in the source pattern definition is not scattered into multiple rules. This improves the modifiability of transformation code. We also have to check the consistency of only one rule with the source pattern definition when we implement and update the transformation. In our change scenario, we only check if the updated `AgentServiceInteraction` rule is consistent with the source pattern definition.

Figure 13 shows this mapping in the whole transformation for semantic web-enabled agent system development.
rule AgentServiceInteraction {
  from (...) <- source pattern for AgentServiceInteraction to (...) 
}

rule OntologyMediation {
  from (...) <- source pattern for OntologyMediation to (...) 
}

rule ContentInterpretation {
  from (...) <- source pattern for ContentInterpretation to (...) 
}

rule AgentRegistration {
  from (...) <- source pattern for AgentRegistration to (...) 
}

... n {
  from (...) <- source pattern for n to (...) 
}

Figure 13 The Mapping of Patterns to Rules in Tefkat

The pattern AgentServiceInteraction is one of the complex source patterns in the transformation. There are other patterns like OntologyMediation, ContentInterpretation, and AgentRegistration. All these patterns can be considered as single concepts and each of them is mapped to a transformation rule written in Tefkat that supports operation composition.

Table 1 summarizes modifiability and rule consistency of the implementations of the case study in two languages. Table rows represent the criteria and table columns correspond to the languages.

<table>
<thead>
<tr>
<th></th>
<th>ATL</th>
<th>Tefkat</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>The transformation for the complex source pattern is decomposed into multiple rules and helpers.</td>
<td>The transformation for the complex source pattern is implemented in a single transformation rule.</td>
</tr>
<tr>
<td>Modifiability</td>
<td>Reduced modifiability due to the decomposition in multiple constructs. All rules and helpers should be updated even for a minor constraint change in source pattern definition.</td>
<td>When a change occurs in the complex source pattern only one rule needs to be modified for this change. This increases the modifiability of the transformation definition.</td>
</tr>
<tr>
<td>Consistency</td>
<td>Since all helpers for the same pattern constrain the same source pattern, the helpers must be kept consistent with each other.</td>
<td>Since there is only one implementation of the constraints of the complex pattern in one rule, there is no need to check the consistency of multiple rules for the same pattern.</td>
</tr>
</tbody>
</table>
The transformation of the complex pattern was decomposed into six transformation rules and ten helpers in ATL; the same transformation was implemented in one transformation rule in Tefkat. The support for the features of complex source patterns (selection of collections, multiple source elements, global filters and selection on the basis of property values) in Tefkat reduced the number of transformation rules and pattern rules for our case. When a change occurs in the complex source pattern, only one rule needs to be modified in Tefkat. This increases the modifiability of the transformation definition.

As we mentioned before, decomposing the transformation for the source pattern into multiple rules and additional structures makes changes in the source pattern scattered over these rules and additional structures. There is no explicit grouping of the decomposed constructs that indicate their relation to the intended source pattern. In the first implementation, modifiability of the transformation code is reduced due to the repetition of code across multiple modules. In ATL, multiple rules and helpers for the same source pattern reduce the modifiability of the transformation code.

Rule consistency and modifiability of transformation code are related. For instance, in ATL, it is necessary to ensure that helpers for the same source pattern are still consistent when updating them for a change in the source pattern. Since there is only one implementation of the constraints of the complex source pattern in one rule in Tefkat, there is no need to check the consistency of multiple rules for the same pattern.

5 Requirements and Mechanisms for Transformation Specifications

In the previous section, we implemented a complex source pattern transformation for a semantic web-enabled agent system as a case study in two well-known model transformation languages, ATL and Tefkat. The rule composition mechanism in ATL requires decomposition of the transformation into multiple rules and additional structures termed helpers. These extra structures used for composing rules reduce the aforementioned qualities of transformation definitions. At a conceptual level, we identify a pattern that forms a complex concept in order to avoid the deficiencies of these rule composition techniques. We map this concept to a single module provided by Tefkat. In this way, the mapping avoids the repetition of source pattern definition in several rules. This criterion is based on the rules and principles defined by Meyer [22] in his study on modularity. Meyer requires that the modular structure of a software solution must be compatible with the model of the problem domain for which the solution is built. This is known as the principle of direct mapping. Another relevant principle is about linguistic modular units: “modules must correspond to syntactic units in the language used” [22]. Our interpretation for direct mapping and linguistic modular units in the model transformation domain is that every source pattern should be able to be transformed by a single rule. The composition of these rules constitutes the entire transformation. Operation composition, a finer grained composition for transformation definitions, is the key feature in this approach. Although some
transformation languages such as Tefkat provide operation composition, the requirements and mechanisms to achieve operation composition are not well-studied.

Section 5.1 explains the requirements transformation languages should fulfil in order to support the principles of direct mapping and linguistic modular units, and why operation composition is needed for this support. Section 5.2 identifies possible model transformation language mechanisms in order to support operation composition and problems about implementing these mechanisms.

5.1 Requirements for Operation Composition

For direct mapping, the query part of transformation languages should first be able to define complex source patterns in a single rule. In our complex source pattern definition, two key issues for complex patterns are; selection of collections (more than one model element is matched by the pattern element, e.g., all the attributes of a class are extracted in a single structure) and multiple source elements. This leads to the following requirement concerning the structure of transformation rules:

Requirement 1: Transformation languages shall support expressing selection of collections and multiple source elements in a single rule with cardinality constraints.

Transformation languages that support these features will be able to express a complex source pattern in a single rule. We have a heuristic rule structure which is an extension of the ATL rule structure for our agent-semantic web service interaction pattern example in Section 3.1.

```
1 rule AgentServiceInteraction {
  2     from ag1: Agent!Agent, [*]pln: Agent!Plan, ag2: Agent!Agent,
  3     role1: Agent!Role, role2: Agent!Role
  4     to agJ1: JAVA!JClass, [*]plnJ: JAVA!JClass, agJ2: JAVA!JClass

1 rule AgentServiceInteraction {
  2     from ag1: Agent!Agent, [1..3]pln: Agent!Plan, ag2: Agent!Agent,
  3     role1: Agent!Role, role2: Agent!Role
```

The symbol ‘[*]’ denotes the selection of a collection for Plan instances in the heuristic rule AgentServiceInteraction. With ag1, pln, ag2, role1, and role2 source pattern elements in the “from” section in line 2 and line 3 of the rule, it defines multiple source pattern elements in its query parts. We shall be able to define cardinality constraints for selection of collections. The transformation language should make it possible to define selection of a specific number of elements in the collection. The following heuristic rule specifies the selection of at least one and at most three Plan instances in a single match of the source pattern definition.

```
1 rule AgentServiceInteraction {
  2     from ag1: Agent!Agent, [1..3]pln: Agent!Plan, ag2: Agent!Agent,
  3     role1: Agent!Role, role2: Agent!Role
```
Specifying selection of collections for more than one pattern element related to each other in the source pattern also causes some ambiguities in matching of the source pattern in the model. For instance, we assume that the metamodel for Agent and Semantic Web Service Interaction in Figure 4 allows many-to-many relation between Agent and Interface classes. The following heuristic rule AgentInterface2JClass matches three Agent instances related to six Interface instances in one match:

```java
1 rule AgentInterface2JClass {
4 }
```

The application of the source pattern part of the rule AgentInterface2JClass is ambiguous because there may be two different types of matches when the source pattern is searched on the source models. The cardinalities of the pattern elements are not enough to express the matches of the pattern elements in the source model. For instance, in the relation of the model elements matched by the pattern element ag and the model elements matched by the pattern element intfc, one Agent instance might be related to only two Interface instances or every Agent instance might be related to every six Interface instances. Transformation languages should also support cardinality constraints for relations between the matched instances.

Although the pattern language makes it possible to specify multiple pattern elements in the source pattern definition, the heuristics rules above do not have any constraint definition support in order to relate these pattern elements. Therefore, the support for Requirement 1 requires support for another feature in transformation languages: a filtering (constraints) based on property values of the model being transformed. The following requirement is specified concerning these filters:

**Requirement 2**: Transformation languages shall support a filtering (constraints) based on property values of the model being transformed for multiple source elements in a single rule.

For instance, the rule AgentServiceInteraction should contain constraints defined in a language independent notation in Section 3.1. This will avoid the constraint repetition in different rules, and additional rule structures like helpers and pattern rules. The heuristic rule AgentServiceInteraction with these filters is shown below:

```java
1 rule AgentServiceInteraction {
2     from agl: Agent!Agent, [*]pln: Agent!Plan, ag2: Agent!Agent
3            role1: Agent!Role, role2: Agent!Role {
4         ag1.type = "Semantic Web Agent"
5     and role1.type = "Domain Role"
6         ag2.type = "Semantic Service Matchmaker Agent"
7     and pln.type = "Semantic Service Executor Plan"
8         role2.type = "Registry Role" and ag1.plays = role1
9     and ag2.plays = role2 and ag1.applies = pln
10        and ag2.applies = pln }
```
There is only one pattern element that specifies the selection of collections ([*]pln: Agent!Plan) in line 2 of rule AgentServiceInteraction. There may be cases in which there is more than one pattern element specifying selection of collections. In these cases, there may be some ambiguities in matching these pattern elements. We do not go into detail here, but transformation languages should support some features to avoid them. The constraint part of the rule between line 4 and line 10 contains the filters for the agent-semantic web service interaction pattern.

The support for these requirements creates an ambiguity for creating or manipulating target elements. There is a match for source model elements and target model elements for the rule AgentServiceInteraction below (we skip the matches for role1 and role2):

Source Model: {ag1: Agent}, {pln: Plan, Plan}, {ag2: Agent}

Target Model: {agJ1: JClass}, {plnJ: JClass}, {agJ2: JClass}

The match for the source part has one Agent instance for the ag1 source pattern element (Semantic Web Agent), two Plan instances for the pln source pattern element (Semantic Service Executor Plan), and one Agent instance for the ag2 source pattern element (Semantic Service Matchmaker Agent). For this match, we can create one JClass instance for the agJ1 target pattern element (Java class for Semantic Web Agent) and one JClass instance for the agJ2 target pattern element (Java class for Semantic Service Matchmaker Agent). The symbol "[*]" denotes the creation of a collection for JClass instances for Plan elements. It just specifies that there may be more than one JClass instance for plnJ target pattern element. Since it does not specify how many JClass instances should be created for a match in the source pattern, there is an ambiguity in creating and manipulating collections in the target part. The rule may create one JClass instance for the plnJ target pattern element. In this case, there will only be one JClass instance in the target model for every two Plan instances in the source model. This match is shown below:

Source Model: {ag1: Agent}, {pln: Plan, Plan}, {ag2: Agent}

Target Model: {agJ1: JClass}, {plnJ: JClass}, {agJ2: JClass}

Therefore, we need the direct links we named transformation operations between source and target pattern elements in a single rule for transformation definitions. They may also define different atomic operations in the rule, like add, update, and delete within the context of transformation. Transformation languages should provide some mechanisms to compose these operations in a single rule, like rule composition mechanisms that compose rules within a single transformation. This leads to the following requirement concerning the composition mechanisms at the operation level:
Requirements: Transformation languages shall support defining operations between source and target pattern elements and composition granularity to compose these rules in a single rule.

The extension of the heuristic rule `AgentServiceInteraction` with direct links between source and target pattern elements is shown below:

```java
rule AgentServiceInteraction {
    from ag1: Agent!Agent, [*]pln: Agent!Plan, ag2: Agent!Agent
    role1: Agent!Role, role2: Agent!Role
    to agJ1: JAVA!JClass, [*]plnJ: JAVA!JClass, agJ2: JAVA!JClass
    action {
        op1:{
            type: add,
            source: ag1, target:agJ1},
        op2:{
            type: add,
            source: pln, target:plnJ},
        op3:{
            type: add,
            source: ag2, target:agJ2}
    }
}
```

The action part of the rule between line 6 and line 9 has three operations: `op1`, `op2`, and `op3`. They are adding operations (every operation has a type attribute which may be set to add, delete, or update) and create target model elements. Every operation has a source and target parts. They can be used to link source and target pattern elements. They avoid the ambiguity in creation and manipulation of target model elements because they specify which target element is created or manipulated for which source element. For instance, the operation `op2` in line 8 specifies an add operation. It creates one JClass instance in the `plnJ` target pattern element for every Plan instance matched by the `pln` source pattern element in the source part. There is a match for source model elements and target model elements for the extended rule `AgentServiceInteraction` below (we skip the matches for `role1` and `role2`):

Source Model: `{ag1: Agent}, {pln: Plan, Plan}, {ag2: Agent}

Op1  ↓  Op2  ↓  Op3  ↓


Operations defined between source and target elements avoid ambiguities in the target part for creating JClass instances for Plan instances in the source model. This action part in the rule structure requires a composition granularity in operation level. Transformation languages should support mechanisms to compose multiple operations to constitute a transformation rule.

5.2 Mechanisms for Operation Composition

In this section, we have discussed possible mechanisms and features for operation composition listed as follows:
Implicit rule invocation at the operation level

Mapping cardinality constraints for source and target model elements in operation definitions

Matching cardinality constraints for operations in a single rule.

There may be different composition mechanisms for operations in a single rule. One possible mechanism for operation composition is to apply implicit rule invocation at the operation level.

In ATL, every rule has an assignment in the ‘to’ section of the rule structure to set values of target model elements. For instance, accessing the Plan elements related to Agent instances (associatedClass<ag.executorPlans) in line 6 of the rule SemanticWebAgent2Agent in Section 5.1 is an implicit call of the rule ExecutionPlan2ExecutionPlan. If we group assignments of target elements under the action part of the rule structure, we can use these assignments as an implicit operation invocation within the rule. The extension of the heuristic rule AgentServiceInteraction with assignments of target model elements in the action part is shown below:

```
1 rule AgentServiceInteraction {
2    from ag1: Agent!Agent, [*]pln: Agent!Plan, ag2: Agent!Agent
3    role1: Agent!Role, role2: Agent!Role

4    (// constraints based on property values of source elements)
5    to agJ1: JAVA!JClass, [*]plnJ: JAVA!JClass, agJ2: JAVA!JClass

6    action {
7        op1:{type:add, source:ag1, target:agJ1}
8          (name< ag1.type,
9           associatedClass<ag1.applies),
10        op2:{type:add, source:pln, target:plnJ}
11          (name<pln.name + '_' + pln.type),
12        op3:{type:add, source:ag2, target:agJ2}
13          (associatedClass<ag2.applies)
14    )
15 }
```

The heuristic rule AgentServiceInteraction given above corresponds to the single Tefkat rule AgentServiceInteraction in Section 4.2. The MAKE part of the Tefkat rule is the implementation of the action part of the heuristic rule. Every operation has its own assignment in this rule structure. The operation op1 sets the attributes of the JClass instance created by agJ1 to the values of Agent instance matched by ag1 in line 8 and line 9. The assignment (associatedClass<ag1.applies) in line 9 can be used as an implicit operation invocation to call the operation op2. Before assigning the association between the JClass instance for Semantic Web Agent and the JClass instance for Semantic Service Executor Plan in the operation op1, JClass instances for Semantic Service Executor Plan should be created in operation op2. A transformation engine should resolve whether this call is an operation call or rule call because this assignment may also call a rule that creates JClass instances for Plan instances instead of operation op2.

We can add more expressivity to the action part but this may create other problems in operation composition. For instance, one issue that could be added to the operation is the cardinalities of source and target model elements in operations. One JClass
instance is created in \( plnJ \) for one Plan instance matched by \( pln \) unless we specify cardinality for these instances in the operation \( op2 \). The mappings in the operations \( op1 \), \( op2 \) and \( op3 \) are one-to-one. There may be cases, however, in which one-to-many, many-to-many, and many-to-one mappings should be defined for source and target pattern elements matching collection of elements. Assume we want to create two JClass instances in the target model for every one Plan instances in the source model. One JClass will be assigned as Semantic Service Executor Plan JClass; one JClass will be assigned as Semantic Helper Plan JClass. We can add one more operation, \( op4 \), to the rule \( AgentServiceInteraction \):

```java
1 rule AgentServiceInteraction {
2   from ag1: Agent!Agent, [*]pln: Agent!Plan, ag2: Agent!Agent
3     role1: Agent!Role, role2: Agent!Role
4     ( // constraints based on property values of source elements)
5   to agJ1: JAVA!JClass, [*]plnJ: JAVA!JClass, agJ2: JAVA!JClass
6     action {
7       op1:{type:add, source:ag1, target:agJ1}
8         (name<- ag1.type,
9          associatedClass<-ag1.applies),
10      op2:{type:add, source:pln, target:plnJ}
11         (name<-pln.name + '_' + pln.type),
12      op3:{type:add, source:ag2, target:agJ2}
13         (associatedClass<-ag2.applies)
14      op4:{type:add, source:pln, target:plnJ}
15         (name<-pln.name + '_' + 'Semantic Helper Plan'),
16    }
17  }
```

The operations \( op2 \) and \( op4 \) denote a one-to-many mapping between Plan and JClass elements. The action part in the rule \( AgentServiceInteraction \) above is not expressive enough to specify this kind of mapping. Therefore, we composed two operations, \( op2 \) and \( op4 \), to define this mapping. The operation \( op2 \) creates one JClass instance for Semantic Service Executor Plan JClass; the operation \( op4 \) creates one JClass instance for Semantic Helper Plan JClass. We can improve the expressivity of the heuristic rule structure to define this kind of mapping in one operation with mapping cardinalities:

```java
1 rule AgentServiceInteraction {
2   from ag1: Agent!Agent, [*]pln: Agent!Plan, ag2: Agent!Agent
3     role1: Agent!Role, role2: Agent!Role
4     ( // constraints based on property values of source elements)
5   to agJ1: JAVA!JClass, [*]plnJ: JAVA!JClass, agJ2: JAVA!JClass
6     action {
7       op1:{type:add, source:ag1, target:agJ1}
8         (name<- ag1.type,
9          associatedClass<-ag1.applies),
10      op2:{type:add, source:[1]pln, target:[2]plnJ}
11         (plnj[1].name<-pln[1].name + '_' + pln[1].type,
12         plnj[2].name<-pln[1].name + '_' + 'Semantic Helper Plan'),
13      op3:{type:add, source:ag2, target:agJ2}
14    }
```
We specify a one-to-many mapping between Plan and JClass elements with mapping cardinalities \((\text{op2}):(\text{type}:+, \text{source}:[1] \text{pln}, \text{target}:[2] \text{plnJ})\) in line 10. The mapping cardinalities \(([1] \text{pln}, [2] \text{plnJ})\) in line 10 specify that the operation \(\text{op2}\) creates two JPlan instances in the target model for each Plan instance matched by \(\text{pln}\) pattern element in the source model. Accessing the target model elements in the assignment part of the operation \(\text{op2}\) in line 10 may be a problem because assignments for two JClass instances should be defined separately. We have to distinguish which JClass will be assigned as Semantic Helper Plan and which JClass will be assigned as Semantic Service Executor Plan. Therefore, we use the array notation from programming languages in order to access the elements of the collection. For instance, we assign the values to the name attribute of the first \(\text{plnJ}\) instance by using \((\text{plnj}[1].\text{name})\) in the collection in line 11.

There may be a need for more expressivity in specifying mapping cardinalities for operation definitions. A fixed cardinality pattern \([1]\) might be specified in the source pattern definition part of the transformation rule. For instance, the rule \textit{AgentServiceInteraction} below specifies a selection of collection for Plan instances with a fixed cardinality constraint \(([3] \text{pln}: \text{Agent!Plan})\) in its source pattern part. In this rule, we want to match three Plan instances in one match and create five JClass instances in the target model. But an additional constraint for this creation is that four of the five JClass instances should be created from two of the three Plan instances and one of the five JClass instances should be created from one of the three Plan instances. In order to implement this constraint in the operation part of the rule, we should add matching cardinality constraints other than mapping constraints for operation definitions to the heuristic rule \textit{AgentServiceInteraction}:

```java
tuple AgentServiceInteraction {
    role1: Agent!Role, role2: Agent!Role
  to agJ1: JAVA!JClass, *[1] plnJ1: JAVA!JClass,
     *[1] plnJ2: JAVA!JClass, agJ2: JAVA!JClass
  action {
    op1:(\text{type}:+, \text{source}:ag1, \text{target}:agJ1)
      (\text{name}:- ag1.type,
       \text{associatedClass}:- ag1.applies),
    op2:(\text{type}:+, \text{source}:[1] \text{pln}(2), \text{target}:[2] \text{plnJ1})
      (\text{plnj}[1].\text{name}:- \text{pln}[1].\text{name} + '_' + \text{pln}[1].\text{type},
       \text{plnj}[2].\text{name}:- \text{pln}[1].\text{name} + '_' + 'Semantic Helper Plan'),
    op3:(\text{type}:+, \text{source}:[1] \text{pln}(1), \text{target}:[1] \text{plnJ2})
      (\text{plnj2}.\text{name}:- \text{pln}.\text{name} + '_' + 'Registration Plan'),
    op4:(\text{type}:+, \text{source}:ag2, \text{target}:agJ2)
      (\text{associatedClass}:- ag2.applies)
  }
}
```
We specify the matching cardinality constraints for operations op2 and op3 in the rule AgentServiceInteraction. The operation op2 creates two JClass instances for each of two Plan instances (source:[1]pln(2), target:[2]plnJ1) in line 11. ‘[1]’ in the source statement ([1]pln(2)) and ‘[2]’ in the target statement ([2]plnJ1) are the mapping constraints and specify that op2 creates two JClass instances in the target model for one Plan instance in the source model. ‘(2)’ in the source statement ([1]pln(2)) is the matching cardinality constraint for op2 and specifies for how many matched Plan instances the operation op2 will be invoked. Since it is obvious that there will be four JClass instances created by op2, we do not have to specify any matching constraint for the target part of op2. Figure 14 shows the source and target matches with operation cardinalities for the rule AgentServiceInteraction.

|------------------|---------------------------------------------------------------|

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Source Match</td>
<td><img src="source" alt="Source Match Diagram" />(2)</td>
<td><img src="source" alt="Source Match Diagram" />(1)</td>
</tr>
<tr>
<td>Target Match</td>
<td><img src="target" alt="Target Match Diagram" />(2)</td>
<td><img src="target" alt="Target Match Diagram" />(1)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Target Match</td>
<td><img src="target" alt="Target Match Diagram" />(2)</td>
<td><img src="target" alt="Target Match Diagram" />(1)</td>
</tr>
</tbody>
</table>

Figure 14 Source and Target Matches with Operation Cardinalities

When implementing a model transformation language with operation composition support, the right balance between language expressivity and language implementation complexity has to be found. Support for new features like mapping cardinalities may require additional features in transformation languages. Furthermore, the complete support for operation composition in transformation languages may cause some problems in transformation execution. Especially, more expressivity in the operation composition may create additional complexity in transformation algorithms. Again, this requires finding the right balance between different qualities: more expressive constructs cost more difficult implementation and inefficiency.
6 Discussion

In the previous section, we explained the requirements transformation languages should support for a better decomposition of transformation and why operation composition is needed for this support and gave a possible model transformation language mechanism for Operation Composition. In this section, we discuss a proposal for a metamodel for model transformation languages supporting the operation composition mechanism given in Section 5.2 and future work to implement this metamodel.

According to the rule mechanism given in Section 5.2, illustrated with the heuristic rule AgentServiceInteraction that supports operation composition in its action part, a transformation language should support the following: expressing selection of collections and multiple source elements in a single rule, filtering (constraints) based on property values of the model being transformed for multiple source elements in a single rule, and operations between source and target elements and composition granularity to compose these rules in a single rule. The concept diagram of the transformation specification with Operation Composition in Figure 3 gives a basis for these features. Based on the concept diagram in Figure 3 and these features given in Section 5, Figure 15 proposes a transformation language metamodel that includes Operation Composition.

In Figure 15, the TransformationSpecification metaclass specifying the transformation composes the TransformationRule metaclass. This composition relation specifies the rule composition granularity in the transformation specification. A single rule contains pattern elements that are represented by the PatternElement metaclass in the proposed transformation language metamodel.

Figure 15. Transformation Metamodel for Operation Composition
With the one-to-many link between the TransformationRule and PatternElement metaclasses, the language metamodel gives support to the requirement for multiple source elements in source and target pattern definitions. Unlike the concept diagram in Figure 3, we do not have a metaclass specifying the source and target patterns. The attribute sourcePatternElement which has a Boolean type specifies whether the element is a source pattern element or a target pattern element. The PatternElement metaclass has other attributes: lower and upper. The attribute lower denotes the minimum number of elements that should be matched by the pattern element; the attribute upper denotes the maximum number of elements that could be matched by the pattern element. These attributes can be set to values like \((0,1), (0,*), (1,*)\) where the symbol ‘*’ denotes the variable cardinality for the pattern element. Therefore, the metamodel with the attributes upper and lower of the PatternElement metaclass supports the selection of collections in pattern definitions. The links between pattern elements are specified by instances of the ElementRelation metaclass in the metamodel. Pattern elements have constraints that can be expressed with OCL expressions. The PatternElement metaclass has the MetaConstraint metaclass. This MetaConstraint metaclass has the OCLExpression metaclass derived from the OCL metamodel [27]. Therefore, pattern elements specified by the PatternElement metaclass instances may have filtering (constraints) based on property values of the model element instances.

To define operations under the action part of the transformation rule, the TransformationRule metaclass composes the Action metaclass with the TOperation metaclass. The composition link between the Action metaclass and the TOperation metaclass specifies the second composition granularity (operation composition) for the transformation specification. The TOperation metaclass defines operation links between source and target pattern elements and is connected to these elements with the TOperationEnd metaclass. Different operations, like create, read, update and delete are specified in the operationType attribute of the TOperation metaclass. To specify the cardinalities of the source and target pattern element instances in the operations, the TOperationEnd metaclass has the upper and lower attributes. For instance, the cardinalities \([1]\) for pln and \([2]\) for plnJ in line 10 of the rule AgentServiceInteraction in Section 5.2 denotes that the operation op2 \{op2: \{type: add, source: [1]pln, target: [2]plnJ\}\} creates two JClass instances in plnJ for every Plan instances in pln. Figure 16 gives a part of the instance model of the last version of the rule AgentServiceInteraction in Section 5.2 for the operation op2 with operation cardinalities. This instance model is a realization of the metamodel given in Figure 15.
The \textit{op2} instance of the \textit{TOperation} metaclass has two instances named \textit{plnEnd} and \textit{plnJEnd} of the \textit{TOperationEnd} metaclass to relate the instances \textit{pln} and \textit{plnJ} of the \textit{PatternElement} metaclass. The lower and upper attributes of the \textit{plnEnd} and \textit{plnJEnd} instances specify the cardinalities of the operation \textit{op2}. Therefore, this operation creates two model elements matched by the \textit{plnJ} pattern element in the target model for every model element matched by the \textit{pln} pattern element in the source model.

One future dimension for work on operation composition is the realization of the transformation language metamodel given in Figure 15. First, we need an abstraction mechanism to specify the prerequisites of operation composition like defining filters, multiple pattern elements in a single rule, and selection of collections for these pattern elements. The concept of role \cite{7} \cite{14} is an ideal candidate for this abstraction. The motivation behind our choice is that the properties of a role model already support these prerequisites of operation composition. Although there are numerous works on using roles in object-oriented data models, we focus on model roles that are played by model elements. A model role has a base metaclass in the metamodel of the model and this role is played by the model element in this model. The role modelling technique allows developers to express domain-specific design patterns as a sub-language of the modelling language, and also allows us to map the model elements into metaclass instances. This technique serves as a bridge between the metalevels in the metamodel hierarchy.

By using role models, we can define multiple pattern elements with their cardinalities and filters (metalevel constraints). Figure 17 shows an example of a \textit{Plan Role} with its cardinality and filter.
The filtering for pattern elements can be specified as *metalevel constraints* which are the well-formedness rules for the base metaclasses of the model elements characterized by the role. These metamodel constraints are expressed using Object Constraint Language (OCL) [27]. <<Plan Role>> in Figure 17 depicts Plan instances in an Agent model whose base class is the *Plan* metaclass in the Agent metamodel. It has a metalevel constraint which shows that the *type* of Plan instance playing this role is a Semantic Service Executor Plan. We can specify the number of instances matched by this role as *realization multiplicity*. A realization multiplicity makes it possible to specify selection of collections for pattern elements. The realization multiplicity of the role named *Pln* in Figure 17 is \{1..*\}. It means that at least one *Plan* instance should be matched by *Pln* role and more than one *Plan* instance can be matched (selection of collection for Plan instances under role *Pln*). To define multiple pattern elements in the source pattern, we can define other roles related to the *Pln* role in Figure 17. Figure 18 gives the definition of the source pattern for Semantic Service Executor Plan and Semantic Web Agent.

The role model in Figure 18 defines another role for Agent instances. <<Agent Role>> specifies Agent instances in an Agent model whose base class is the *Agent* metaclass in the Agent metamodel. It has a metalevel constraint which shows that the *type* of Agent instances is Semantic Web Agent. The realization multiplicity of the *Agnt* role is \{1..1\}. It means that only one *Agent* instance should be matched by the *Agnt* role.

Role modeling [7] [14] gives a well-defined grammar and structure to express complex source patterns including multiple pattern elements, selection of collections, and filtering. A model transformation language based on role modelling may satisfy the prerequisites of operation composition especially within the context of source element selection. One possible problem, as we discussed in the previous section, is the right balance between language expressivity and language implementation complexity. Implementing these features offered by role modelling and other operation composition features proposed in Section 5 may create additional requirements and problems in language implementation.

![Figure 18. Roles for Semantic Service Executor Plan and Semantic Web Agent](image-url)
7 Related Work

In this section, we survey some of the model transformation languages and approaches related to operation composition.

We have experienced the capabilities of ATL [9] and Tefkat [23] within the context of operation composition. ATL 2004 allows only one pattern element in the source pattern definition part of rules. Therefore, we can only define one-to-one and one-to-many transformations in a single rule. Since it does not support specifying multiple source pattern elements and selection of collections in a single rule, complex source pattern definitions and the operation composition in a single rule are not provided by ATL 2004. ATL 2006 allows multiple source elements and has expressivity closer to the one provided by Tefkat, but it still does not support selection of collections in a single rule. Therefore, there is no need for operation composition in both ATL versions.

In Tefkat, the source part of rules makes it possible to define more than one pattern element and constraints on these elements. The language does support selection of collections in the source part but it does not make it possible to specify cardinality constraints for these collections. The ability to define more than one source pattern element and to select collections requires the operation composition support found in Tefkat. But Tefkat does not support some advanced features such as mapping cardinalities in operation definitions.

Epsilon Transformation Language (ETL) [16] is the language implemented in the context of Epsilon which provides an integrated set of languages for common model management tasks. Since a rule defines a source (following the \texttt{transform} keyword) and many target (following the \texttt{to} keyword) parameters in ETL, ETL has a similar mechanism as ATL to handle complex source patterns. However, it is quite feasible to build a transformation language supporting complex patterns in Epsilon platform, as it provides a base object navigation (query) language, on which you can build new languages. Another framework to build a transformation language for complex patterns is OpenArchitectureWare (OaW) [29] which supports aspects of model-driven software development such as parsing, validation and transformation of models.

MOF QVT 2.0 [28] makes it possible to define more than one pattern element and constraint on these elements, but does not support selection of collections in the source part. The ability to define more than one source pattern element reduces the number of rules in the example implementation (in contrast to the ATL implementation). However, the lack of collection selection still prevents us from capturing the source pattern into a single rule.

GMORPH [25] [26] is one of the model transformation languages considering operation composition in a single rule. It attempted to package generative and graph transformational techniques into a single coherent language. Sendall describes some drawbacks [26] for combining these two powerful techniques. For instance, the relationship between the Left Hand Side (LHS)/ Right Hand Side (RHS) graphs and the rule script can be clouded by the differences in notations and styles, requiring careful consideration of the LHS/RHS structures when writing the rule script. As such, there is a need for a rich set of checks to ensure that the two views are consistent and well-formed. To use a more declarative approach, for instance mapping and the symmetry
between LHS and RHS graphs, one could relate each pattern element in the LHS to its corresponding RHS element. This relies upon the symmetry in the two sides to allow one to deduce the mapping strategy at the instance level [26]. The symmetry between source and target pattern elements is provided by direct links between these elements. The language also provides a cardinality construct for pattern elements to support selection of collections for them. Although the GMORPH language considers pattern element cardinalities and operation definitions in its rule structure, they do not give any usage context for complex source pattern transformations. There is also no sign of the implementation and realization of the language. The author does not give any implementation level issues, such as problems in pattern matching for multiple pattern elements and selection of collections on models, but the language concept is very close to the operation composition definition we describe in this paper.

MOLA [10] is another language that supports direct operations between source and target pattern elements. There are links that relate source elements to corresponding target elements. Although there are some attempts to define operation links and compose them under a rule structure in GMORPH and MOLA, there is no clear definition to address complex source pattern transformation problems within the context of operation composition. They do not mention the balance between language expressivity and language implementation complexity.

GReAT [1] is a graph-based model transformation language that supports defining multiple operations in its rule structure. It has three distinct parts: pattern specification, graph transformation and control flow language. The pattern specification language is used to define the source and target patterns of the transformation specification in Figure 3; the graph transformation language is used to define transformation rules with their action parts. Transformation rules written in the pattern specification and graph transformation parts are composed in the control flow language which specifies execution order over these rules. GReAT allows querying and transforming multi element patterns in a single rule but it does not support selection of collections in pattern definitions. It only supports simple pattern matching. A simple pattern is the exact pattern that is being searched for in a larger structure [1]. Therefore, the operations in GReAT are based on one-to-one mappings. Specifying one-to-many or many-to-one transformations in operations in a single rule is not possible in GReAT. Matching only simple patterns in the source pattern limits the ability to specify and compose operations in GReAT. Unlike in the proposed model transformation language given in Section 6, it is not possible to define direct operations between source and pattern elements and specify cardinalities in both operation ends in GReAT. Although matching algorithms are given for pattern definitions which support selection of collections in [1], the current version (1.6.0) of GReAT does not support it. As we mentioned in Section 5.1, there will be a need for operations between source and target pattern elements to avoid ambiguities in the target model when the implementation of GReAT supports all features of complex source patterns like selection of collections.

The issue of modifiability of transformation code is also addressed from an aspect-oriented perspective. Since the same constraint might be applied in many different places in a transformation, it would be beneficial to describe a common constraint in a modular manner [19] [20]. Lengyel et al. [19] [20] discuss the problem of the cross-cutting constraints in graph transformation steps and provide an aspect-oriented solu-
tion (the weaving algorithm used to propagate aspect-oriented constraints to graph transformation steps) for crosscutting constraints. The main difference between this work and our paper is that they assume there are constraints that are applied in different parts of the transformation. Their approach for modularization of transformation constraints can be applied to reduce the problems we encounter in the sample implementation in ATL. In the example, we have ten helpers that implement the same constraints for different context types. The repeated constraints can be modularized as a constraint aspect and this constraint aspect could be woven to the helpers. On the other hand, poor transformation language support to separation of concerns in the transformation code causes these problems. In this paper, we study the language support for a better separation of concerns in transformation definitions by using operation composition.

8 Conclusion

In this paper, we discussed the need for a finer grained composition at the operation level supported by transformation language mechanisms for complex source pattern transformations. As a case study, we implemented a complex pattern transformation for a semantic web-enabled agent system in order to show the deficiencies of current rule composition mechanisms in transformation languages. We used two well-known model transformation languages, ATL and Tefkat.

There are different interpretations of the term “complex” for source patterns. Therefore, we identified some features in order to classify a pattern as complex: multiple source elements, selection of collections, global filters, and selection on the basis of a property value from the model being transformed. A simple pattern is one which does not use at least one of these features. Current composition techniques provide proper support for composition of rules with simple source patterns. There are cases in which complex structures have to be selected. Transformation composition at the rule composition level may not provide proper support to define complex source pattern transformations in these cases.

Our example in ATL showed that the implementation of complex source pattern transformations at the rule composition level required decomposition of the transformation into multiple rules and additional structures. The decomposition of the transformation for a complex source pattern in ATL reduced certain qualities of the transformation specification, such as modifiability of transformation code and transformation rule consistency.

At a conceptual level, we identified a “complex source pattern” that has to be expressed as distinct constructs in concrete transformation languages. In order to map this concept to a single language construct, every complex source pattern should be transformed by a single rule. This requires the support of the features we identified for complex source patterns based on operation composition in order to avoid the ambiguities for transforming elements in the action part of the rule. The key feature in this solution is operation composition, a finer grained composition for model transformations.
We mapped this concept to a single module provided by Tefkat. The support for the features of complex source patterns and the composition at the operation level in Tefkat reduces the number of transformation rules required for complex source patterns. This improves solutions provided by current rule composition techniques. Since every complex source pattern is handled in a single rule, the changes in the source pattern are not scattered over the whole transformation specification. This improves the modifiability of transformation code. There are no multiple rules or additional structures that define the constraints for the same pattern in the transformation code. Therefore, there is no need to check the consistency of rules for the same source pattern. This has a positive effect on the rule consistency of transformation.

Although some transformation languages such as Tefkat provide operation composition, the requirements and mechanisms to achieve operation composition were not well-studied in the context of model transformation language construction. However, operation composition requires additional integration mechanisms in model transformation languages.

When implementing a solution in a given language, the developer has to find the optimum balance between decomposition of rules and using large complex rules. Clearly, it is not feasible to define the entire transformation in a single rule. There will be some cases where decomposition of the transformation is essential to the management of complexity. Support for new features for operation composition may require additional features in the transformation languages. Furthermore, the required support for complex patterns in transformation languages may cause some problems in transformation execution. In particular, more expressivity in the source pattern definitions may create additional complexity in pattern matching algorithms. Again, this requires striking of the optimum balance between different qualities: more expressive constructs could be more difficult to implement and eventually be more inefficient. On the other hand, a less expressive transformation language may reduce the modifiability of the transformation code.

References


