CAViT: a Consistency Maintenance Framework based on Transformation Contracts

Pieter Van Gorp, Dirk Janssens

Formal Techniques in Software Engineering, Universiteit Antwerpen,
{pieter.vangorp,dirk.janssens}@ua.ac.be

Abstract. Model driven engineering is a software engineering methodology that aims to manage the complexity of frameworks by relying on models and transformations. Unfortunately it is only poorly understood where and how this new methodology differs from traditional methodologies. Therefore, this paper formalizes how contract based model transformation extends existing design by contract theory. The key extension is that transformation contracts can be maintained automatically by mapping consistency invariants to the postconditions of transformation rules. When an invariant is violated, the corresponding transformation rule will be called provided that its precondition is satisfied. This paper presents the *Contract Aware Visual Transformation* (CAViT) framework, an implementation of declarative middleware for contract based model transformation. We illustrate how CAViT can be used to integrate UML based visual model transformations with OCL based transformation contracts.

Introduction

Integrating packages and applications written in a variety of languages is a major challenge for the software engineering community [1]. Luckily, programming languages and their associated libraries have gone a long way. Thanks to modern integration platforms such as J2EE [2] one can, for example, connect mainstream ERP packages such as SAP [3] with legacy applications written in COBOL and a reasoning engine written in PROLOG. Unfortunately, this flexibility comes at the cost of high complexity. Managing entities and processes that are scattered over all applications is problematic when there is a lack of documentation on how all applications relate to the overall requirements. Even when such documentation is in place, it is often unfeasible to update all sources and documentation affected by a particular business change without advanced tool support. This paper formalizes the mechanics of model-driven engineering tools (1) by defining how the languages used to program such tools relate to object oriented design by contract and controlled graph rewriting and (2) by describing their architecture. Section 1 introduces the reader to design by contract and model-driven engineering, section 2 presents domain specific models of a sample application, section 3 explains how these models can be kept consistent using a new contract aware visual model transformation approach and section 4 describes the architecture of the tool that validates this approach. Section 5 relates this contribution to related work and section 6 concludes this paper.

Dagstuhl Seminar Proceedings 05161
Transformation Techniques in Software Engineering
http://drops.dagstuhl.de/opus/volltexte/2006/429

1 Supporting Technologies

This section first describes two methodologies that were succesfully applied to tackle the complexity of large scale software development: design by contract (DBC) and model driven engineering (MDE). In the context of MDE, we describe “contract based model transformation”, a new application of the design by contract foundations for maintaining the consistency between related models. Finally, this sections covers Model Driven Architecture (MDA) and visual model transformation.

1.1 Design by Contract

DBC is a software correctness methodology for procedural and object-oriented software. It relies on logical assertions to detect implementation mistakes at run-time or to prove the absence thereof at compile-time.

The fundamentals of DBC were developed by Floyd and Hoare in the late sixties [4,5]. By formalizing the effect of programming language constructs on the state of variables in axioms and inference rules, Hoare illustrated the feasibility of proving program correctness. The proposed proof systems are based on *state assertions*, which are logical expressions about the values of program variables. These state assertions are used to state that a program S will ensure a state assertion q (called the *postcondition*) provided that state assertion p (called the *precondition*) holds right before it is executed, or $\{p\}S\{q\}$. A correctness proof consists of a deductive sequence of state assertions and axioms or inference rules from precondition to postcondition.

In the early seventies, Hoare and Wirth published a proof system for the programming language PASCAL [6]. Meanwhile, verfication condition generators based on backward substitution were built to automatically derive proof obligations [7]. Finding what rules form the shortest path from precondition to postcondition remains a creative activity but a theorem proover can help discharging proof obligations given a library with the required axioms and inference rules. Today, one can rely on industial-strength proof assistants with inference rules for object-oriented languages [8].

DBC has also found its way into incomplete verification methods. In this context, the precondition and postcondition are checked at test execution time. For this purpose, Meyer included support for expressing assertions in the Eiffel programming language [9] while Kramer built iContract for extracting assertions from Java comments [10]. The advantage of the testing approach is that it is applicable even when complete coverage is unfeasible. On the other hand, deviations between the contracts and the implementation may find their way to the production environment due to an incomplete set of test cases.

1.2 Model Driven Engineering

MDE is a methodology that proposes to tackle the complexity of software development by treating models and transformations as first-class software artifacts [11].

1.2.1 Terminology By collecting the mainstream definitions for the core MDE concepts, this section can be used as an introduction and refererence to the MDE terminology adopted in this paper.

Definition 1. *A model is a simplified representation of a part of the world named the system [12].*

Studies in human-computer interation distinguish between conceptual and mental models [13]. The former refer to representations made by a teacher (or designer) to explain a system to a user (or maintainer). A mental model on the other hand is the system representation that a user (or maintainer) actually applies in his mind when working with (or on) the system. In MDE, one proposes to map *models* in their conceptual sense to source code by means of transformations [11]. MDE is a relevant methodology in those cases where source code is too far from the mental models of the designers and maintainers of the software. It allows these stakeholders to work and communicate with models that match their minds more closely.

Kleppe et al. define a transformation as "*the automatic generation of a target model from a source model, according to a transformation definition*" [14]. We generalize this definition in order to support transformations that are carried out manually:

Definition 2. *A transformation is the construction of a set of target models from a set of source models. A transformation is automatic if it can be applied mechanically according to a transformation definition.*

Since model transformation is the heart and soul of MDE [15] it is not restricted to particular modeling languages, compilers or execution platforms. Rather than enforcing one language for cognitive modeling (like the UML [16]) it embraces the variety of languages and libraries that need to be reconciled in heterogenous software architectures. The fundamental difference with Rapid Application Development (RAD [17]) is that transformation definitions can be created and/or optimized by software architects.

Since a model transformation is intended to manipulate a set of models one can call it a *metaprogram*. The structure of the input and output modeling languages is represented by so-called *metamodels* while a *transformation definition* consists of (1) references to the set of models as data and (2) transformation rules as behavior. Based on Favre's observations on metamodels [18] we define:

Definition 3. *A metamodel is a structural model of a language. It consists of (1) a type system (classes with attributes and associations with cardinalities) for the abstract syntax elements and (2) well-formedness rules that capture syntactic validity constraints not covered by (1).*

Type system modeling can be done with either UML class diagrams or Entity Relationship (E/R [19]) diagrams but fundamentally one tends to rely on graph based models with attributed nodes and edges. In the graph transformation community, the type system of a modeling language can be encoded in a *type graph*. Type graphs cannot enforce that for example "the name of a class should be unique within its package". Such *well-formedness rules* need to be enforced with additional graph constraints, some of which cannot be represented visually. In the context of object-oriented metamodeling, they correspond to the invariants of metaclasses.

Kleppe et al. define a transformation definition as a set of transformation rules that together describe how a model in the target language can be constructed from a model in the source language [14]. We generalize this definition by considering transformation definitions that manage any non-zero number of models. Moreover, we do not restrict a model to be used as in- or output exclusively:

Definition 4. *A transformation definition contains (1) a set of models conforming to metamodels and (2) a set of transformation rules that together describe how its models can be constructed from one another.*

Defining model transformations in a general purpose programming language such as Java lies quite far from the conceptual task of model (graph) transformation. In *Reflective MDE* one defines model transformations by means of models themselves [20]. Such transformation models abstract from the complexities and evolution of model management frameworks such as the NetBeans MetaData Repository (MDR [21]) or the Eclipse Modeling Framework (EMF [22]) [23]. Section 1.2.3 will illustrate that transformation models expressed in a visual modeling language based on the UML and graph transformation appear to match people's mental models of the domain very well.

Definition 5. *A transformation rule is a description of how one or more constructs in one model can be constructed from one or more constructs in another model.*

Again, this definition is based on the work of Kleppe et al. [14]. Note that these authors promote to represent transformation rules as objects. This enables one to store the source-target relationships of a transformation in the object state of the rule that created the target from the source. Instead, we propose to represent *transformation definitions as objects*. This design choice was motivated by our intention to represent models and transformations with ordinary object-oriented data structures. Nevertheless, there are MDE applications that need to reason explicitly about the source-target relationships (so-called *traceability links*) between models and model elements that were created from one another. The following definition allows one to describe where this information is stored:

Definition 6. *A traceability model consists of relations between models and/or model elements that are generated from one another by one or more model transformations.*

1.2.2 Contract based Model Transformation Contract based model transformation is a new application of the design by contract foundations targeted at maintaining the consistency between related models.

Definition 7. *A transformation contract is a pair of constraints (called the pre- and postcondition) that describe the effect of a transformation rule on the set of models contained in its transformation definition. The postcondition describes the model consistency states that the rule can establish provided that its precondition is satisfied. The postcondition of a transformation rule corresponds to an invariant of the transformation definition in which it is contained.*

The notion of transformation contracts can be used to formalize the notion of incremental consistency maintenance:

Definition 8. *A transformation contract of a rule can be maintained automatically by calling the rule (1) as soon as the invariant corresponding to its postcondition is violated and (2) provided that its precondition is satisfied.*

Note that in model inconsistency states where the preconditions of more than one rule are satisfied, transformation engines are allowed to call the rules in arbitrary order. Some model inconsistency states cannot be resolved automatically. These states do not satisfy the precondition of any transformation rule whose postcondition describes a consistent model state. For simple contracts, one can automatically generate the satisfying transformation rules. On the other hand, complex cases require the manual development of transformation rules. For example, in the context of the WODN case study [24] we considered constraints that asserted that for a particular match in the source model, there should be an element in the target model that shares particular properties with some elements from the source model. A simple violation scenario (or rule *precondition*) is the case where one has developed an initial version of the source model and the target model has not been generated yet. A transformation engine can automatically instantiate a target model and populate it with a model element that satisfies the constraint.

Contract aware transformation engines should support the creation of *traceability links* between model elements that are created from one another. Such links should be accessible in a model transformation language that integrates with the constraint checking language. This enables the manual development of model transformations for violations occurring in models that were previously generated from one another. Generating transformation rules for such "*co-evolution conflicts*" is beyond the state-of-the-art of today's transformation engines and perhaps even unfeasible since there are a lot of ways to reconcile existing models automatically and expert knowledge is required to decide what manual (external) changes to the models should be preserved or overriden. As noted before, Kleppe et al. proposed to include traceability models inside transformation rules but they did not describe how the traceability information could be used for model reconciliation.

1.2.3 Visual Model Transformation Writing transformations manually can be facilitated by using specialized transformation languages (such as YATL [25]) since they hide the algorithmic details of navigating and manipulating the object graphs representing the models. YATL is a textual transformation language that has been succesfully applied to a number of MDE case studies. However, it is relatively young compared to the languages developed in the domain of graph rewriting. In this domain there is a general consensus that representing transformations visually makes them most understandable to humans [26]. Therefore, in visual model transformation, one applies the foundations of graph rewriting to model driven engineering by interpreting models as *graphs* and metamodels as *type graphs* [23]. The fundamental research topics raised by the new application domain are transformation contracts (see Definition 7) and traceability models (see Definition 6).

1.2.4 Model Driven Architecture Model Driven Architecture (MDA) is an OMG initiative to develop a set of standards for integrating MDE tools [27]. In order to enable more stakeholders to exchange models in a language that matches their mental modeling language to an acceptable degree, the UML was extended with a range of new syntax constructs [28].

MDA would not be a standard for MDE if it would concern only one modeling language. Therefore, it defines a language called MOF which is responsible for defining modeling languages. It is based on an extension of UML class diagrams (see Fig. 1) and OCL (see Fig. 2). The former should be used to model the structure of a metamodel whereas the latter should be used to encode its well-formedness rules. More and more UML tools [29,30] and data warehousing tools [31] provide a MOF based interface to their repository, either message-based with Java (based on the JMI standard [32]) or document-based with XML (based on the XMI standard [33]).

Although model transformations can already be implemented by using these languages, the OMG has also recognized the need for a dedicated MOF transformation language [34]. OCL based transformation contracts fully comply with the OMG's request for a declarative transformation language. A MOF framework that would allow one to integrate model transformations written manually in imperative transformation languages (such as YATL) would enable transformation writers to compensate for the limitations of today's OCL based transformation engines [24].

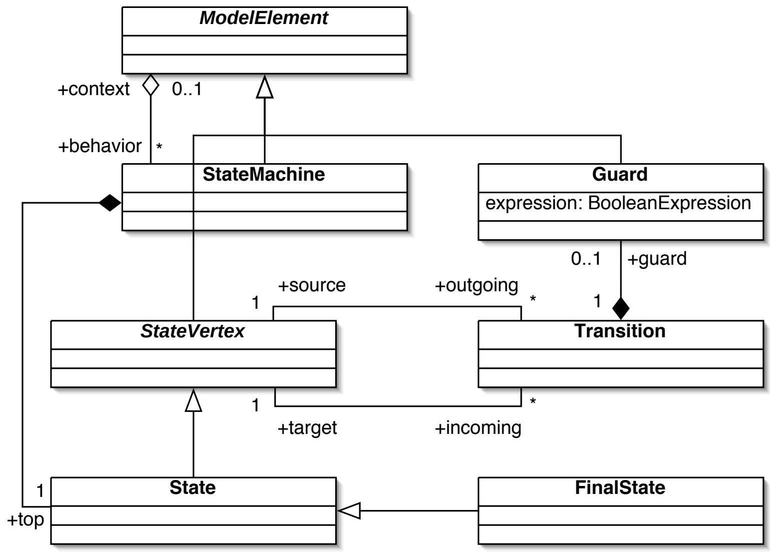

Fig. 1. MOF diagram displaying an excerpt of the statemachine part of UML 1.5. A statemachine is a model element that represents the *behavior* of another *context* model element. A statemachine contains exactly one *top* state that is connected to other states by transitions (*incoming* and *outgoing*) that can contain a *guard*.

```
context FinalState:
        self.outgoing->size = 0
```

Fig. 2. OCL Well-formedness rule from the UML metamodel stating that a final state should never have outgoing transitions.

2 Models of a Meeting Scheduler

To give a realistic idea of models that need to be kept consistent, this section presents some domain specific models of a sample application. The application is based on the *Meeting Scheduler* problem statement that was proposed by Van Lamsweerde et al. [35] as a benchmark for requirements elicitation and software specification techniques. The problem statement of the benchmark was published deliberately imprecise and incomplete [36]. The first part of the problem statement reads as follows:

> *Meetings are typically arranged in the following way. A meeting initiator asks all potential meeting attendees for the following information based on their personal agenda:*
> – *a set of dates on which they cannot attend the meeting (hereafter referred as exclusion set);*
> – *a set of dates on which they would prefer the meeting to take place (hereafter referred as preference set).*
>
> *A meeting date is defined by a pair (calendar date, time period). The exclusion and preference sets are contained in some time interval prescribed by the meeting initiator (hereafter referred as date range).*
> *The initiator also asks active participants to provide any special equipment requirements on the meeting location (e.g., overhead-projector, workstation, network connection, telephones, etc.). He/she may also ask important participants to state preferences about the meeting location.*

Sections 2.1, 2.2 and 2.3 present some illustrative *conceptual*, *robustness* and *physical data* models repectively. The fragments in this paper should merely illustrate some realistic dependencies between models in different languages and should not be regarded as a complete nor stable specification of a meeting scheduler.

2.1 Conceptual Models

Fig. 3 displays a part of the conceptual model stating that the application distinguishes between personal and professional meetings. Both types of meetings will be planned according to different rules. We define this taxonomy of meeting schedules using the concept *view inheritance* [37]. The concrete type of a schedule (being either *Professional Schedule* or *Personal Schedule*) is determined dynamically based on the value of the *type* attribute (called the *discriminator*) of the Schedule class. View inheritance allows conceptual modelers to define taxonomies in multiple dimensions without (1) having to define artificial classes that apply multiple inheritance for all possible combinations of subclasses from the different taxonomies or (2) restricting the number of inheritance lattices to one and using the strategy pattern [38] for encoding the other dimensions.

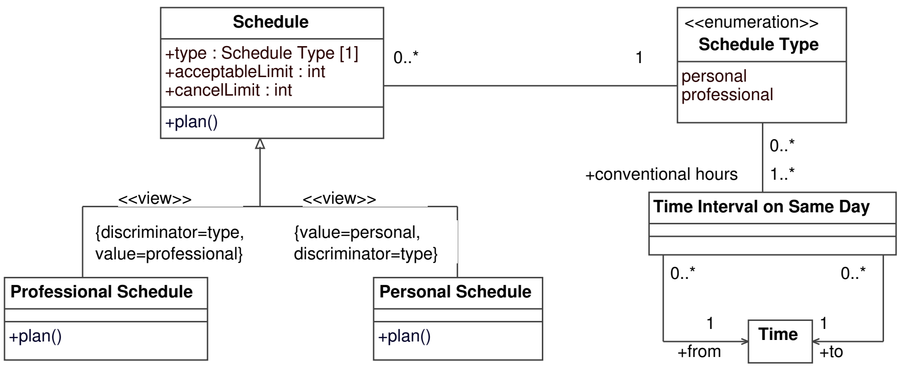

Fig. 3. Data structure for representing the planning dimension (or *view*).

Fig. 4 is a diagram modeling that schedules are made for a set of meeting attendees each of whom are related to a particular user of type *Person*. Both persons and meeting locations have an address. Attendees (especially those who will give a presentation) can use their *required equipment* bag to indicate what equipment should be present at the meeting location. The planning algorithm can then use the association between *Meeting Location* and *Equipment Copy* to check what locations provide all the required equipment.

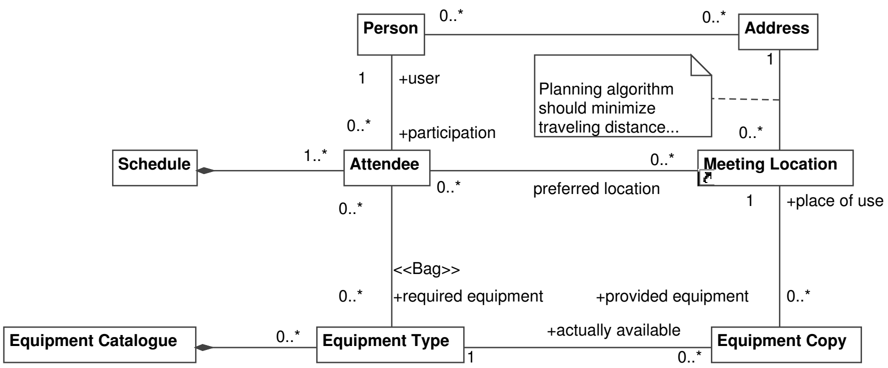

Fig. 4. Data structure for representing the location preferences of meeting attendees.

Fig. 5 displays that each attendee can specify (1) in what time intervals he/she cannot attend, (2) in what time intervals he/she would prefer to attend and (3) the flexibility (or priority) for (2). Allowing the use of association classes (such as *Flexibility*) allows the conceptual modeler to represent the relationship for date preferences directly between its major participants (*Attendee* and *Time Interval*) without putting too much focus on the auxiliary *Flexibility* class. The *acceptableLimit* and *cancelLimit* attributes of class *Schedule* serve as deadline mechanisms in the algorithm for proposing an optimal meeting location and date.

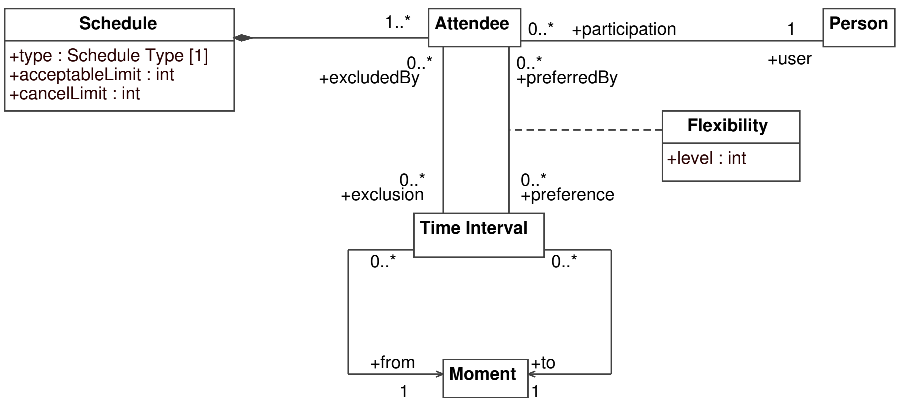

Fig. 5. Data structure for representing the date preferences of meeting attendees.

2.2 Robustness Models

As a second example of domain specific models, we present a demonstrative robustness diagram of the meeting scheduler. The notation applied in Fig. 6 is an extension to UML proposed by Rosenberg and Scott [39] and promotes a clear separation between user interface, controllers (services) and persistent data (entities). Each Entity in a robustness model corresponds to a class in a conceptual model. The diagram on Fig. 6 describes the flow of events in the "Confirm Meeting" use case: after logging in, a meeting initiator can click on a *Confirm Meeting* button on the main screen of the application. This triggers a call to the service for managing the status of a meeting. This *Status Service* component updates the meeting status on the *Schedule* entity and arranges a reservation for the required meeting location. Finally, all meeting attendees are notified by mail.

2.3 Physical Data Models

Fig. 7 shows the physical data model corresponding to the conceptual model from Fig. 5. The model conforms to a profile for physical data modeling [40] based on

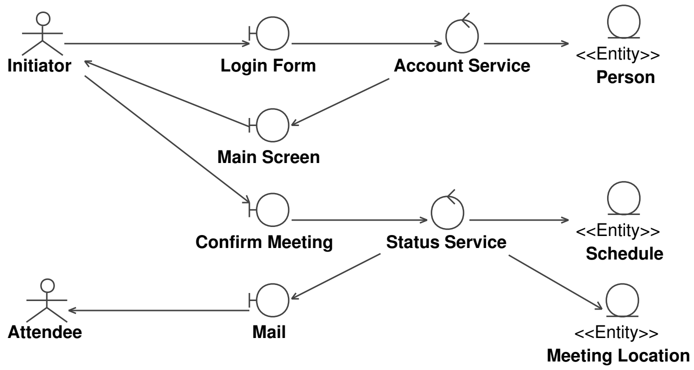

Fig. 6. Robustness Diagram of the “Confirm Meeting” use case. Actors are shown on the left, followed by interfaces and services in the middle and entities on the right.

the work of Ambler [41]. The mapping from classes and associations with association classes to tables, associative tables, keys and foreign keys corresponds to the mapping from entity sets and relations with attributes to relational schemas as covered in introductory database course books [42].

Each class from the logical data model is mapped to a table. All attributes are included as columns, along with extra columns for realizing the associations between the classes. The physical data model contains explicit nodes for keys and foreign keys such that all relations are defined precisely.

The associative table *ATTENDEE-TIMEINTERVAL* maps to the association class *Flexibility.* The table maps to two outgoing one-to-many associations. We present this mapping problem to illustrate that there is a consistency relation between conceptual models and physical data models demanding that each class maps to a table and vice versa. Büttner and Gogolla have described how the flattened structure can be *constructed* from the model with association classes [43] but their rules need to be plugged into a framework where a constraints are *checked* permanently. If, for example, a database administrator migrates all data from the *TIMEINTERVAL* table to another *T_RANGE* table, then a constraint should fail after the former table is dropped. This violation could be resolved by modifying the constraint or updating the traceability model.

3 Metamodels and Model Transformations

The meta-information of the domain specific models presented in the previous section is defined on the same “meta-level” (see [18]) as the application models of the meeting scheduler. This *language profile* technique is based on decorating model elements with special flags (called *stereotypes*) and properties (called *tagged values*). Neither the method of contract based model transformation, nor its implementation in CAViT is

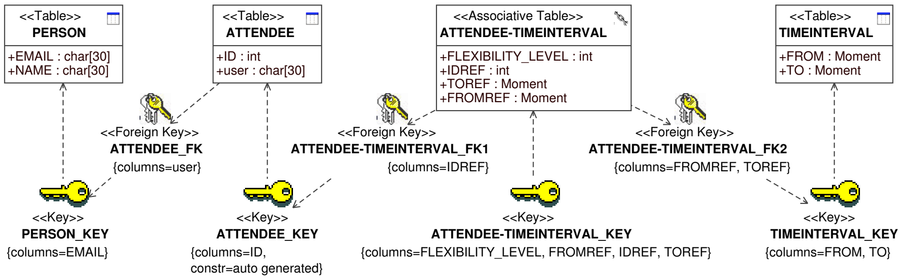

Fig. 7. Physical data model of “date preferences” information.

restricted to the language profile technique. Therefore, the reader is not supposed to master all details of this technique. Nevertheless, the overview given in section 3.1 may be useful for understanding the running example of this paper completely. Section 3.2 shows the UML profiles used in this example. Finally, section 3.3 presents the definition of an illustrative model transformation.

3.1 Language Profiles

Definition 9. *A stereotype is a model element that models (or “represents in a simplified manner”) a subtype of an existing metaclass, called the “base class”, when “base class” is part of a metamodel whose type system should be fixed.*

Fixing the type system of a language’s metamodel is needed when models that are expressed in that language *need to be exchanged with a standard API or data structure*. Since our domain specific models are represented naturally in UML editors, we can benefit from the portability of our models across UML tools at the cost of the complexity introduced by metamodeling through the language profiling technique.

When the type system of the modeling language is fixed, extra meta-information needs to be expressed in the language itself. In the context of object-oriented metamodeling, a stereotype is a model element A that is an instance of a metaclass B that has a “name” attribute (instantiated to “A” in this example) and that is associated with the metaclass C part of metamodel D. Although conceptually A is a metaclass that subclasses C, it is impossible to create instances of A in repositories that are based on the types of D. Instead, one can state that a model element E that is an instance of type C conforms to A by adding A to the set of Bs associated with E.

Example stereotypes from Fig. 7 are *Table*, *Foreign Key*, *Key* and *Associative Table*. In an object-oriented metamodeling environment, these model elements (corresponding to “A”) are instances of the UML metaclass *Stereotype* (corresponding to “B”). All four stereotypes are defined on UML metaclass *Classifier* (corresponding to “C”) of metamodel *UML* (corresponding to “D”). *Table*, *Foreign Key*, *Key* and *Associative Table* conceptually subclass *Classifier* but technically they are only associated to *Classifier* through the *base class* attribute of UML metaclass *Stereotype*. Fig. 8 elaborates on the design of profiles in the UML metamodel. The definition of the stereotypes for the physical data modeling profile is visualized on Fig. 9. On Fig. 7, *Table* is applied to model element Person (corresponding to “E”).

Definition 10. *A tagged value is a model element that models an attribute of a stereotype.*

Example tagged values from Fig. 7 are *columns* and *constr* which are defined on Fig. 9.

Stereotypes and tagged values are distributed in packages, called *profiles*, that can be included as libraries in applications. Since stereotypes and tagged values are model elements in their own right, they have to be formalized in the metamodel of the modeling language being used. Since we have modeled our sample Meeting Scheduler in UML, Fig. 8 displays the part of the MOF metamodel of UML responsible for defining and applying profiles. A model element can have zero or more stereotypes associated with it. The right side of the diagram shows that each stereotype can have a number of contained tag definitions. This enables the composition on the left side of the diagram to be populated: a model element can have many tagged values associated with it. Each such tagged value is either string-based (when the dataValue is instantiated) or conforms to exactly one tag definition (when the association with end *referenceValue* is populated). A tag can associate a model element with multiple values (such as the columns tag on *TIMEINTERVAL_KEY* on Fig. 7), depending on the *multiplicity* property of the tag definition. The language profile technique is also implemented in Java, where tagged values are called *annotations*.

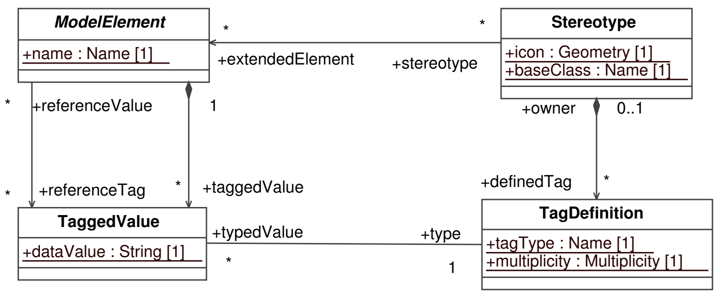

Fig. 8. MOF diagram of UML metamodel support for profiles.

When working with UML profiles, well-formedness rules (WFRs) are expressed in OCL and in the context of a stereotype at the modeling level (M1 in the OMG metalevel architecture [44]). Contrast this with the MOF approach to metamodeling where OCL metamodel WFRs are defined in the context of metaclasses (M2 types in OMG’s metalevel architecture). Practically, this difference in metalevels means that profile WFRs will not be validated by generic MOF transformation tools. Consequently, profile conformance can only be ensured by using an additional M1 constraint checker for UML. Today’s OCL validators [45,46] support both M1 OCL checks on UML models and M2 OCL checks on MOF models.

3.2 Metamodel Definitions

All models presented in section 2 conform to the UML metamodel and apply profiles to obtain a domain specific semantics. Figures 1 and 8 are based on the UML specification [16]. We refer the reader to consult this reference for learning how models, packages, classes, attributes and other UML concepts are represented in MOF. This section will briefly present the definition of a domain specific UML profile applied in the *Meeting Scheduler*.

Fig. 9 shows a part of the content of the package defining the “physical data modeling” profile. The *Physical Data Model* stereotype can be used to indicate that a UML model describes the relational database implementation of an application. *Table*, *Associative Table*, *Foreign Key* and *Key* are stereotypes applied in Fig. 7. Note that all tags are defined in the context of a stereotype. The type of the *columns* tag of the *Foreign Key* and *Key* stereotypes is a nonempty list of *Attribute*. The latter is a metaclass from the UML metamodel. The type of the origin tag of stereotype *Key* is *Key Origin*, an enumeration type that is also contained in the profile definition package.

One can imagine an arbitrary number of well-formedness rules on this language profile. One possible rule would state that a *Classifier* marked with the *Foreign Key* stereotype should have an outgoing dependency link to a *Classifier* marked with the *Key* stereotype.

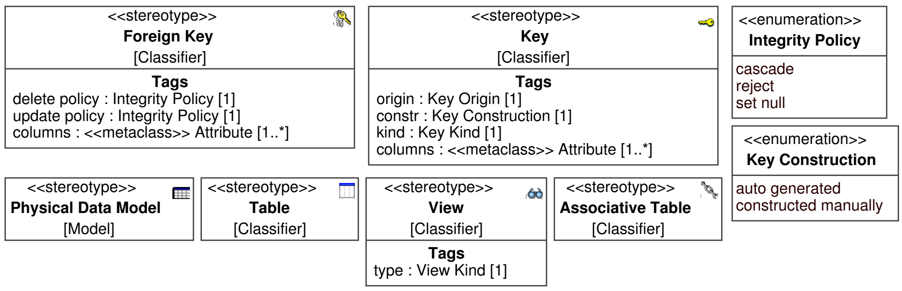

Fig. 9. Part of the UML profile for physical data modeling.

3.3 Transformation Definition

Instead of giving an overview of all transformations that are required to maintain the consistency between conceptual, physical and robustness models, this section provides an in-depth description of one transformation definition.

The example covers the transformation of all classes in the conceptual model to entities in the robustness model. The rationale is that a conceptual model will put all behavior (operations) as closely to the data (attributes) as possible while a robustness model will add to these *entities* a number of *service* facades to encapsultate the flow of control between application screens (called *interfaces* to the *actors*). Note that similarly one could build the reverse transformations to support a methodology where entities are created manually during their discovery in use case refinement based on robustness modeling [39] and then copied to classes in the conceptual model. The focus of this paper lies on presenting a powerful (yet understandable) framework for automating repetitive model management tasks and *not* on introducing a new software engineering process.

The transformation rule is implemented by the method *cmClasses2rmEntities* contained in class *CM2RM*. As Fig. 10 shows, *CM2ES* has three attributes: (1) *applicationModel* denotes the UML model containing applications conforming to the profiles presented in section 3.2, (2) *profileDefinitions* denotes the UML model containing the definitions of the profiles presented in section 3.2, while *applicationName* denotes the name of one application package within *applicationModel* that contains the models that need to be transformed. For our running example, one would use “Meeting Scheduler” as a value for this attribute. The rule *cmClasses2rmEntities* has three parameters: (1) *aModel* corresponds to *CM2ES's* attribute *applicationModel*, (2) *pModel* corresponds to *profileDefinitions,* while (3) *aName* corresponds to *applicationName*. This mapping is specified as part of the transformation contract presented in the following section.

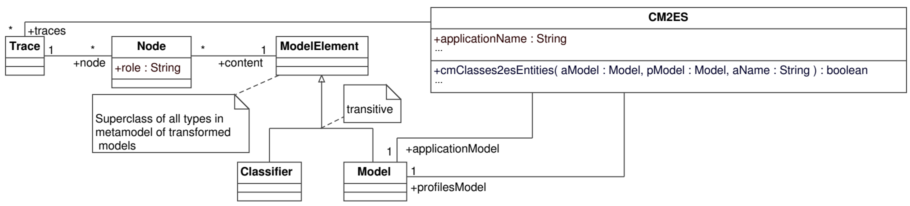

Fig. 10. Structural model of the “conceptual model to robustness model” transformation and its related traceability data.

The *cmClasses2rmEntities* method maintains the transformation contract specified in section 3.3.1 and is implemented with a visual model transformation language as explained in section 3.3.2.

3.3.1 Transformation Contract The consistency relation between conceptual models and robustness models can be ensured by a combination of transformation contracts. Note that in the case of multiple contracts with the same postcondition, their preconditions should be mutually exclusive to ensure that only one rule is called in a particular model inconsistency scenario. The precondition of the contract presented in this paper describes the situation where the robustness model has not been generated yet. The postcondition states that all classes from the conceptual model correspond to entities in an associated robustness model.

Precondition The precondition can be formalized as follows:

```
context CM2ES::pim2psm(aModel:Model, pModel:Model, aName:String): Boolean
pre :
let appPkgs : Package = appFromModel(aModel, aString) in
appPkgs->size=1 and
this.traces->select(appTrace : Trace |
  -- the trace refers to a conceptual model in the application package
  appTrace.node.content->includes(
    appPkgs.ownedElements->(hasStereotype("Conceptual Model"))
  ) and
  -- the trace refers to a robustness model in the application package
  appTrace.node.content->includes(
    appPkgs.ownedElements->(hasSteretype("Robustness Model"))
  )
)->isEmpty
```

This precondition makes use of the “appFromModel” helper operation that looks up the package representing the application with the given name from a UML model:

```
context CM2ES operations:
appFromModel(container: Model, appName: String): Set(Package)=
 allTransParts({container})->select(
   app : Namespace | app.name=applicationName
 ).oclAsType(Package)
```

On the other hand, *appFromModel* calls the “allTransParts” helper that computes the transitive closure of the “ownedElements” association:

```
context CM2ES operations:
allTransParts(s : Set(Namespace)) : Set(Namespace) =
 if s->includesAll(s.ownedElements->asSet) then s
 else allTransParts(s->union(s.ownedElements->asSet))
 endif
```

The precondition also calls the helper “hasStereotype” which can be easily defined on the UML metaclass “ModelElement”. The operation is expected to return whether or not a stereotype with the given name is applied to a model element.

Invariant The sample invariant expresses that within the package representing a particular application, the contained conceptual model should trace to the contained robustness model. Moreover, all classes from the contained conceptual model should trace to a class in a contained robustness model. The latter classes should have the “Entity” stereotype applied to it.

```
context CM2ES: inv:
let a: Package= appFromModel(applicationModel, applicationName) in
 this.traces->includes(t1 |
  t1.node->includes(a.ownedElements->select(hasStereotype("Conceptual Model")
  and
  t1.node->includes(a.ownedElements->select(hasStereotype("Robustness Model")
 ) and
 allClassesFromSubmodel(a, "Conceptual Model")->forAll(cc: Class |
  allClassesFromSubmodel(a, "Robustness Model")->exists(rc: Class |
   this.traces->includes(t2 |
     t2.node->includes(cNode | cNode.content=cc)
     and t2.node->includes(rNode | rNode.content=rc)
   ) and
   cc.name=rc.name and
   rc.hasStereotype("Entity")
  )
 )
```

Postcondition As soon as the *CM2RM* transformation definition is loaded into contract based transformation middleware such as CAViT, its invariants need to be ensured. Fig. 11 shows how the postcondition of *cmClasses2rmEntities* is mapped to our sample invariant.

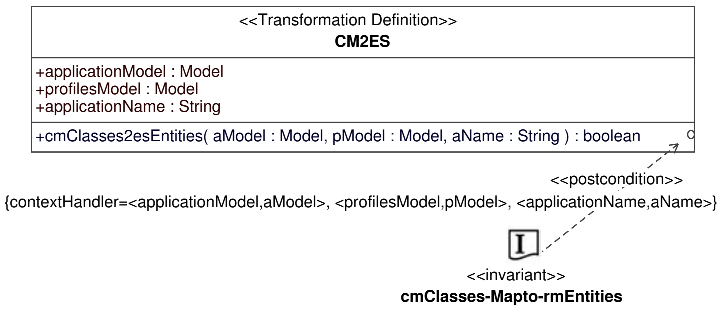

Fig. 11. Declarative configuration of the CAViT framework for managing the CM2ES transformation definition.

3.3.2 Visual Model Transformations This section applies the UML profile for *Story Driven Modeling* (SDM [23]) to implement the *cmClasses2rmEntities* transformation rule conforming to the contract from the previous section. *Story diagrams* are a minimal extension of plain UML (or object orientation) to implement method bodies as visual specifications based on controlled graph rewriting [47]. The language was formalized by Zündorf [48] and first implemented in Fujaba [49], a model-based open source CASE tool. The transformation models presented in this section are input for MoTMoT [50], a MOF compliant re-implementation of (a part of) Fujaba's code generator for story diagrams.

Fig. 12 displays the transformation flow describing the general behavior of transformation rule *cmClasses2rmEntities*. First, the transformation looks up the stereotypes from the conceptual-modeling and robustness-modeling profiles. If a stereotype cannot be found, or any other error occurs in any state, the transformation ends with reporting a failure by returning *false* (which is the name of the first final state). This is indicated by the transitions with the < <*failure*> > stereotype. When no exceptions occur, the transformation goes through four states and ends with reporting succes.

The behavior of the transformation in the first state is further specified in the *lookupStereotypes* primitive that is shown in Fig. 13. The pattern starts from the *profilesAndSamples* node, which represents the UML model that was passed as an argument to the transformation. From there, the pattern navigates to the contained *Profiles* package and binds three stereotypes: one from the *Conceptual Modeling* profile and two from the *Entity Service Modeling* (or *robustness* modeling) profile.

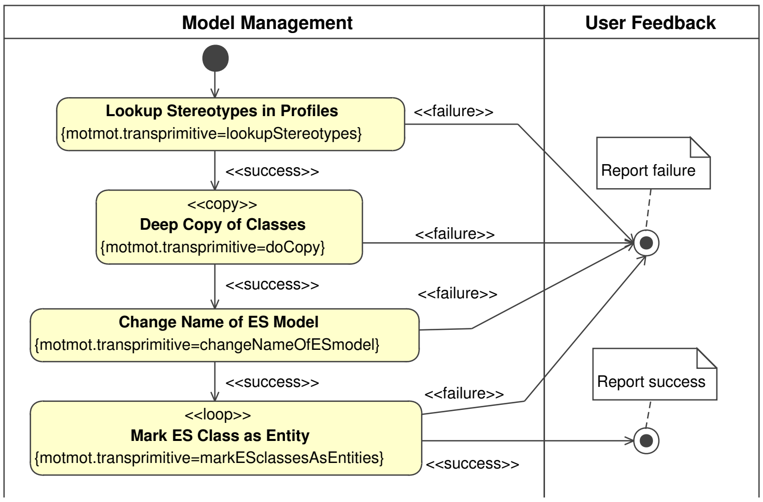

Fig. 12. Flow of the sample transformation process.

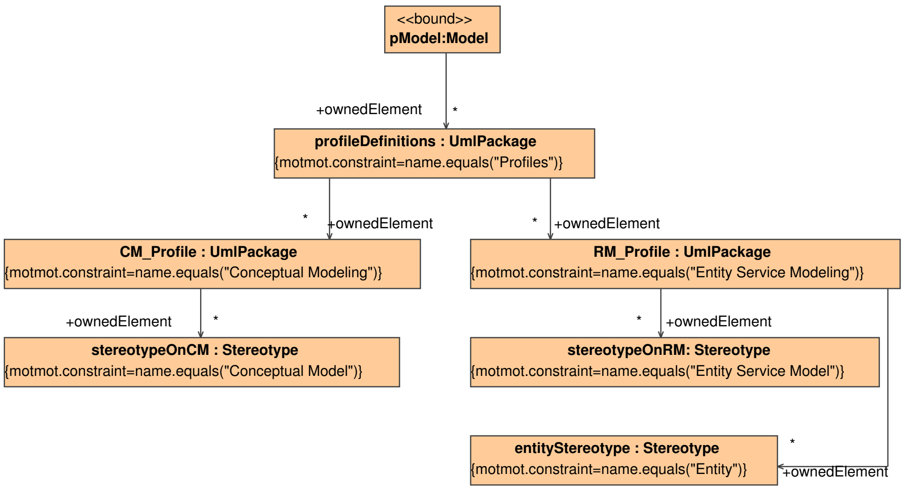

Fig. 13. Specification of the *lookupStereotypes* transformation primitive.

The second transformation primitive is responsible for copying all classes in the conceptual model to entities in the robustness model. Fig. 14 shows the specification of this *"copy"* story.

A "copy story" is a visual language construct responsible for copying subgraphs. If a match of the <<from>> node is found then it is copied to the <<to>> node. Additionally, all nodes and links on a specified composition path starting from the <<from>> node are copied as well. Implicitly, all attributes from a copied node are copied along. Finally, links that connect copied nodes, but that are not part of a specified composition path, can be copied by labeling them with the <<copyRefToCopiedNode>> stereotype.

The *doCopy* primitive shown on Fig. 14 uses the transitive closure construct to find the *Meeting Scheduler* package within all packages that are recursively contained in the overall model. There, it looks for a node *cmModel* of type *Model* that has the <<*Conceptual Model*>> stereotype applied. This node is labeled as <<*from*>> and will be the source of the copy operation. The outgoing composition path indicates that all contained classes and their attributes should be copied. The pattern could be extended to copy the type of the *Attribute* nodes as well. After copying the subgraph to the <<*to*>> node, the latter is added to the *Meeting scheduler* package and the <<*Entity Service Model*>> stereotype is attached.

As shown in Fig. 15, the third story consists of a straightforward attribute assignment on the node representing the robustness model. Its name needs to be changed after it has been created in the previous story since otherwise it would have the same name as the conceptual model.

Finally, the *markRMclassesAsEntities* story shown in Fig. 16 decorates all classes that were copied to *rModel* with the <<Entity>> stereotype.

4 Architecture of the CAViT Framework

This section briefly illustrates how CAViT is related to existing model management software. Fig. 17 shows that CAViT acts as middleware between an OCL based consistency checker (YATL4MDR [51]) and the SDM based model transformer discussed before (MoTMoT [50]).

CAViT is configured by declarative transformation contract specifications that associate violations of OCL consistency constraint with the execution of SDM transformation rules. For example, the CAViT model from Fig. 11 mapped the invariant from section 3.3.1 to the transformation from section 3.3.2.

YATL4MDR and MoTMoT can access the model repository (MDR [21]) through a file-based (XMI [33]) or object-based interface (JMI [32]) interface. CAViT lets them use the latter to eliminate expensive serialization calls. We have extended YATL4MDR with a *Violation Reporter* interface which is based on the *Observer* design pattern. Finally, the interface between MoTMoT and CAViT is defined by the signature of the SDM transformation rules. The parameters of MoTMoT transformation rules contain objects representing elements from the inconsistent models. Behind the scenes, these objects implement JMI based interfaces as well.

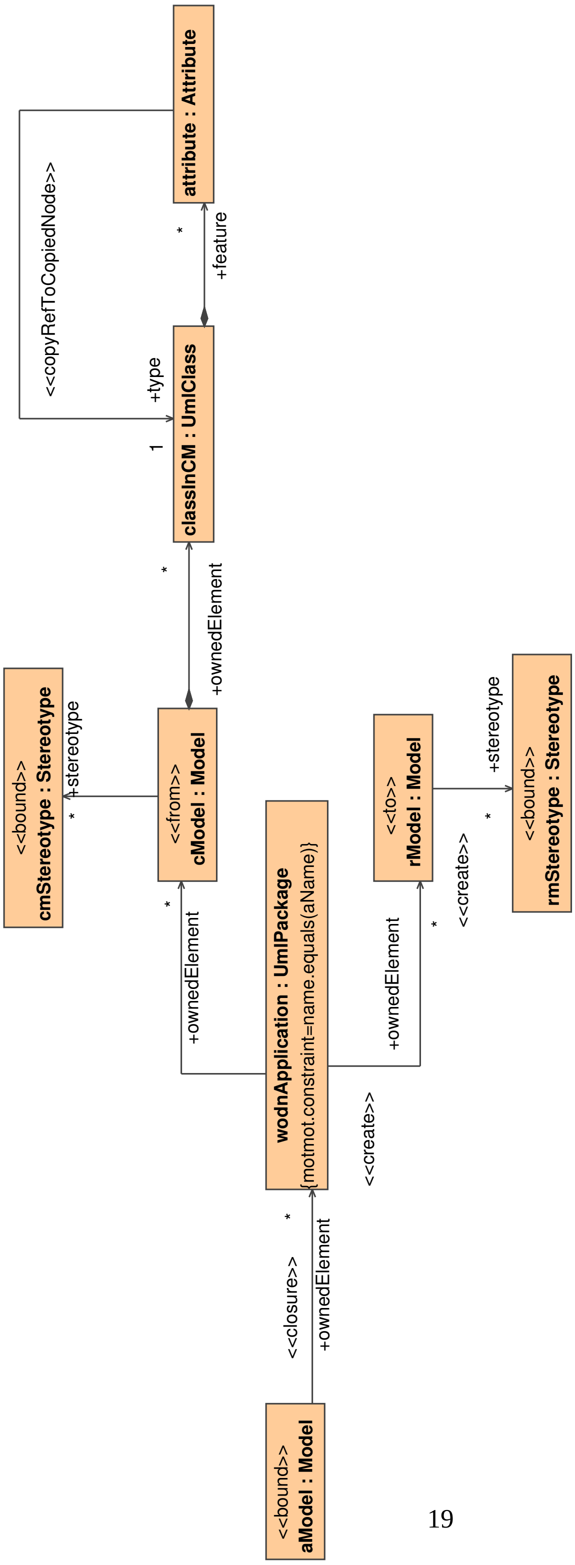

Fig. 14. Specification of the *doCopy* transformation primitive.

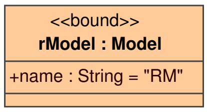

Fig. 15. Specification of the *changeNameOfRModel* transformation primitive.

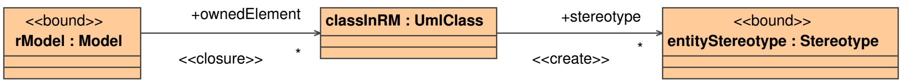

Fig. 16. *Specification of the markRMclassesAsEntities* transformation primitive.

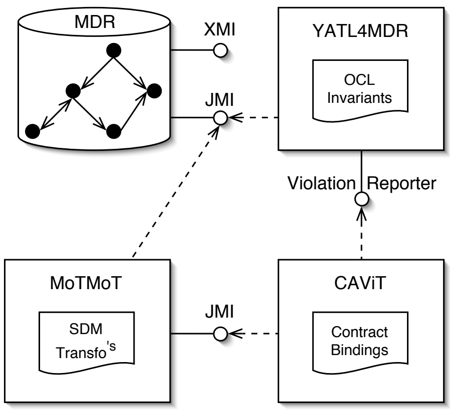

Fig. 17. Architectural diagram of CAViT.

5 Related Work

The CAViT framework bridges two technological spaces. *Object-oriented metamodeling* is used to define the structure and contracts of transformation definitions. The behavior of transformation rules is modeled as controlled graph rewriting rules. Controlled graph rewriting is a well-known concept in the *graph transformation* technological space. By formalizing the technology bridge with mainstream *design by contract* concepts, this paper is related to both technological spaces.

5.1 Object-Oriented Metamodeling

In this technological space, that is promoted by OMG's MDA initiative, OCL was already used to specify the contract of refactorings, which are a specific kind of model transformation [52]. Next to the pre- and postcondition, a *refactoring contract* also contains a constraint that describes a set of code smells. Refactoring contracts can be applied for composing primitive refactorings, verifying the preservation of behavioral properties and automatically removing code smells. Cariou et al. investigated how source and target models could be represented such that the contracts of general model transformations could be represented without superfluous extensions to standard OCL [53]. The authors presented two approaches but did not succeed to define all types of transformations (*inplace* and *outplace* implementations of *rephrasings* and *translations* that are either *horizontal* or *vertical* [54]) without an informal extension to the OCL. Akehurst et al. proposed to model transformation definitions as mathematical relations [55]. A relation maps one metaclass to another one and is defined by:

- a domain and a range specification that states which instances of the source and target metaclass can be mapped onto each-other,
- set-theoretical properties such as bijective, functional, total, etc., and
- relation-specific constraints that consist mainly of equations between attributes of related elements.

All three constraint types are modeled as invariants on a *Relation* class. The second class of constraints is made available automatically to all relations by means of inheritance (from the *Relation* class described in the appendices of [55]). Since in CAViT all consistency constraints are modeled as invariants on classes as well, it is compatible with this framework for contract based model transformation based on set theory. Reasoning about one source and one target model is a special case of the CAViT approach where transformation definitions can hold one, two, or more attribute references to models. Apart from this, the added value of CAViT lies in the definition of behavior for constraint violations: CAViT will delegate to the model transformation rule of which the postcondition is mapped to the failed invariant. By reasoning about invariants, pre- and postconditions, we build upon the well-established foundations of design by contract. Moreover, since CAViT is based on SDM, it provides a UML based syntax to describe the behavior of transformation rules. The latter is a natural extension of the *relational* approach since Akehurst only implemented the bodies of transformation rules in Java due to the absense of precise UML action semantics [56]. Another possibility is the extension of OCL as a side-effect free constraint and query language to a transformation language. Initial experiments were already conducted to build consistent model elements automatically for architectural models [57]. More case studies are required to assess the readability and expressiveness of SDM and the OCL action language in practice. Especially, there's a lack of publications on transformation rules that *reconcile* inconsistent models.

It should be noted that work in the object-oriented metamodeling space builds upon the results of the logic-based knowledge bases. In the latter area, Balzer was among the pioneers that proposed to decouple the definition of declarative consistency contracts from imperative repair actions. Moreover, he recognized the importance of temporarily tolerating consistency violations and the role of manual reconciliation assisted by automatic inconsistency notification [58]. Finkelstein et al. used executable temporal logic [59] to implement transformation rules for maintaining the consistency between software models from different viewpoints [60].

5.2 Graph Transformation

The roots of technological space were developed in the early seventies [61]. The foundations of controlled graph rewriting are described by Schürr in the handbook of graph transformation [26].

Interestingly, one can describe the pre- and postconditions of transformation rules within the graph transformation space itself [62]. Based on these concepts, tools can analyze whether a rule does not break the well-formedness rules of a modeling language [63]. Tom Mens mapped PROLOG-based work on "reuse contracts" to the graph transformation space to illustrate the applicability of graph transformation to the evolution of object-oriented software [64]. Reuse contracts are model transformation rules that were defined to manage the evolution of class hierarchies and collaborations. By defining a set of primitive reuse contracts as graph rewrite rules, Mens was able to derive a conflict matrix. This idea was later elaborated in collaboration with Täntzer and Runge [65]. One should note that the algorithms for computing conflict matrices have not been designed with the control structures and copy operator from SDM in mind.

Since our definition of a transformation contract is not based on a particular constraint language such as the OCL, CAViT's consistency maintenance approach should be applicable to visual specifications of pre- and postconditions as well. This paper complements previous work on pre- and postconditions within the graph transformation space since one only investigated how inconsistency could be prohibited, rather than being tolerated temporarily. Once the work of Mens et al. would be extended to support controlled graph rewriting, it could be used to compute possibly undesired side-effects of executing one transformation rule before another one, when CAViT indicated that both of them could fix a model inconsistency.

As a running example for this paper, we modeled a meeting scheduler system using UML profiles and defined transformations between the different models. We promoted the use of language profiles for prototyping domain specific languages since today's CASE tools allow one to tune general purpose editors to domain specific ones very easily. Nevertheless, we look forward to applying the results from recent advances in editor technology that promise a simple decoupling of abstract syntax models from concrete syntax models. This would significantly simplify our transformation contracts and rules because they would be defined on a dedicated abstract syntax model. Interestingly, it was exactly during the development of such configurable editors that Akehurst et al. proposed to use a contract based model transformation approach as well [66] while Bardohl defines the mapping between abstract and contrete syntax with graph grammars [67].

6 Conclusions

The main theoretical contribution of this paper is that it relates new developments in model transformation technology to existing paradigms. More specifically:

- the notions of transformation definitions and transformation rules were mapped back to classes and methods,
- one or more models are accessible from a transformation definition by storing references to model elements in attributes from the transformation class,
- the notion of consistency contracts was mapped back to class invariants,
- we defined how pre- and postconditions of methods relate to class invariants when automating consistency management, and finally
- the behavior of transformation methods is modeled by executable UML diagrams based on controlled graph rewriting.

Additionally, we illustrated that traceability can be modeled elegantly in an object-oriented fashion: a transformation class is associated with a set of traceability classes. The latter references a set of model elements that are related by a particular traceability relation.

The proposed theory is validated by the implementation and use of the *Contract Aware Visual Transformation* (CAViT) framework. CAViT can be regarded as model transformation middleware since it integrates two model management frameworks that were developed separately:

- YATL4MDR is used to evaluate model consistency contracts that are specified in OCL,
- MoTMoT is used to execute the visual transformation models.

By combining the strengths of object-oriented metamodeling with graph rewriting, CAViT enables the development of more complex transformations and the exploration of new research topics such as metamodel and transformation evolution.

Acknowledgements

The authors would like to thank Hans Schippers and Olaf Muliawan for their contributions to MoTMoT. Hans Schippers also provided valuable feedback to make the UML profiles related to MoTMoT and CAViT as readable as possible. This work has been sponsored by the Belgian national fund for scientific research (FWO) under grants “Foundations of Software Evolution”, “A Formal Foundation for Software Refactoring” and “Formal Support for the Transformation of Software Models”. Other sponsoring was provided by the European research training network “Syntactic and Semantic Integration of Visual Modeling Techniques (SegraVis)”.

References

1. Carlo Ghezzi and Gian Pietro Picco. An outlook on software engineering for modern distributed systems. In *Proceedings of the Monterey workshop on Radical Approaches to Software Engineering*, Venice (Italy), 8 2002.
2. Sun Microsystems. Java 2 platform enterprise edition specification, v1.3, July 2001.
3. SAP. mySAP ERP. http://www.sap.com/solutions/business-suite/erp/, 2005.
4. R. Floyd. Assigning meanings to programs. In J. T. Schwartz, editor, *Proc. Symposium on Mathematical Aspects of Computer Science*, volume 19, pages 19–32. American Mathematical Society, 1967.
5. C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.
6. C. A. R. Hoare and N. Wirth. An axiomatic definition of the programming language pascal. *Acta Informatica*, 2:335–355., 1973.
7. Shigeru Igarashi, Ralph L. London, and David C. Luckham. Automatic program verification i: A logical basis and its implementation. *Acta Informatica*, 4:145–182, 1974.
8. David Crocker. Safe object-oriented software: the verified design-by-contract paradigm. In F.Redmill and T.Anderson, editors, *Proceedings of the Twelfth Safety-Critical Systems Symposium*, pages 19–41. Springer-Verlag, 2004.
9. Bertrand Meyer. *Eiffel, the Language*. Prentice Hall, 1992.
10. R. Kramer. iContractor - The Java Design by Contract Tool. In *Technology of Object-Oriented Languages and Systems (26th International Conference and Exhibition)*, 1998.
11. Jean Bézivin and Olivier Gerbé. Towards a precise definition of the OMG/MDA framework. In *Proc. 16th Int’l Conf. Automated Software Engineering*, page 273. IEEE Computer Society, November 2001.
12. E Seidewitz. What models mean. *IEEE Software*, 20, Sept.-Oct. 2003.
13. Donald Norman. *Mental Models*, chapter Some Observations on Mental Models, pages 7–14. LEA, 1983.
14. Anneke Kleppe, Jos Warmer, and Wim Bast. *MDA explained: the model driven architecture: practice and promise*. Object Technology Series. Addison – Wesley, 2003.
15. Shane Sendall and Wojtek Kozaczynski. Model transformation – the heart and soul of model-driven software development. *IEEE Software, Special Issue on Model Driven Software Development*, 20(5):42–45, 2003.
16. Object Management Group. Unified Modeling Language (UML), March 2003. version 1.5. document ID formal/03-03-01.
17. James Martin. *Rapid application development*. Macmillan Publishing Co., Inc., 1991.
18. Jean-Marie Favre. Foundations of meta-pyramids: Languages vs. metamodels – episode II: Story of thotus the baboon. In Jean Bezivin and Reiko Heckel, editors, *Language Engineering for Model-Driven Software Development*, number 04101 in Dagstuhl Seminar Proceedings. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, 2005.
19. Peter Pin-Shan S. Chen. The entity-relationship model: Toward a unified view of data. *ACM Transactions on Database Systems*, 1(1):9–36, 1976.

20. Jean Bézivin, Nicolas Farcet, Jean-Marc Jézéquel, Benoît Langlois, and Damien Pollet. Reflective Model Driven Engineering. In Perdita Stevens, Jon Whittle, and Grady Booch, editors, *Proc. UML 2003 - The Unified Modeling Language. Model Languages and Applications. 6th International Conference, San Francisco, CA, USA*, volume 2863 of *LNCS*, pages 175–189. Springer, October 2003.
21. Sun Microsystems. NetBeans Metadata Repository, 2002. http://mdr.netbeans.org/.
22. Bill Moore, David Dean, Anna Gerber, Gunnar Wagenknecht, and Philippe Vanderheyden. *Eclipse Development using the Graphical Editing Framework and the Eclipse Modeling Framework*. IBM Redbooks. International Business Machines, January 2004.
23. Hans Schippers, Pieter Van Gorp, and Dirk Janssens. Leveraging UML profiles to generate plugins from visual model transformations. *Software Evolution through Transformations (SETra). Satellite of the 2nd Intl. Conference on Graph Transformation*, 127(3):5–16, 2004.
24. Pieter Van Gorp, Dirk Janssens, and Tracy Gardner. Write once, deploy N: A performance oriented MDA case study. In *Enterprise Distributed Object Computing Conference, (EDOC)*, pages 123–134, 2004.
25. Octavian Patrascoiu. YATL:Yet Another Transformation Language. In *Proceedings of the 1st European MDA Workshop, MDA-IA*, pages 83–90. University of Twente, the Nederlands, January 2004.
26. Grzegorz Rozenberg, editor. *Handbook of graph grammars and computing by graph transformation: volume I. foundations*. World Scientific Publishing Co., Inc., 1997.
27. Object Management Group. Model Driven Architecture (MDA), July 2001.
28. Conrad Bock. UML 2 activity and action models. *Journal of Object Technology*, 2(4), 2003.
29. No Magic. Magicdraw. http://www.magicdraw.com/, 2005.
30. Gentleware. Poseidon for UML, version 2.6, 2005. http://www.gentleware.com.
31. Daniel T. Chang. Cwm enablement showcase: Warehouse metadata interchange made easy using cwm. *TDWI*, 11(Data Warehousing: What Works?), 05 2001.
32. Java Community Process. Java metadata interface (JMI) specification – JSR 000040, June 2002.
33. Object Management Group. XML Metadata Interchange (XMI), v2.0. formal/03-05-02, 2003.
34. Object Management Group. MOF 2.0 Query / Views / Transformations RFP ad/2002-04-10, October 2002.
35. A. v. Lamsweerde and R. Darimont and P. Massonet. The Meeting Scheduler System - Problem Statement. Technical report, Université Catholique de Louvain - Département d'Ingénierie Informatique, B-1348 Louvain-la-Neuve (Belgium), 1992.
36. M.S. Feather, S. Fickas, A. Finkelstein, and A. van Lamsweerde. Requirements and specification exemplars. *Automated Software Engineering*, 4(4):419–438, 1997.
37. Bertrand Meyer. *Object-Oriented Software Construction*, chapter Multiple Criteria and View Inheritance. Prentice Hall, 2nd edition, 1997. http://archive.eiffel.com/doc/manuals/technology/oosc/inheritance-design/section_10.html#HDR39.
38. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*, chapter 5 (Strategy), pages 315–323. Professional Computing Series. Addison-Wesley, 1995.
39. Doug Rosenberg and Kendall Scott. *Use case driven object modeling with UML: a practical approach*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.
40. Pieter Van Gorp. UML Profile for Data Modeling. http://www.fots.ua.ac.be/ pvgorp/research/datamodelingprofile/, 2005.
41. Scott W. Ambler. A UML Profile for Data Modeling. http://www.agiledata.org/essays/umlDataModelingProfile.html, 2005.

42. Jeffrey D. Ullman and Jennifer Widom. *A first course in database systems*, chapter 3.3, pages 103–107. Prentice-Hall, 1997.
43. Fabian Büttner and Martin Gogolla. Realizing UML metamodel transformations with AGG. In Reiko Heckel, editor, *Proc. ETAPS Workshop Graph Transformation and Visual Modeling Techniques (GT-VMT'2004)*, ENTCS. Elsevier, 2004.
44. Object Management Group. *Meta Object Facility (MOF) specification*. Object Management Group, 2002. Version 1.4. Available for download at url http://cgi.omg.org/cgi-bin/doc?formal/2002-04-03.
45. Birgit Demuth. The Dresden OCL Toolkit and its Role in Information Systems Development. In *13th International Conference on Information Systems Development: Methods and Tools, Theory and Practice*, Vilnius (Lithuania), 9 2004.
46. Dan Chiorean. Using OCL Beyond Specification. In *Proceedings of the 4th International Conference on the UML "Modeling Languages, Concepts and Tools"*, number 2185 in Lecture Notes in Computer Science, Toronto (Canada), 2001. Springer-Verlag.
47. Lars Grunske, Leif Geiger, Albert Zündorf, Niels Van Eetvelde, Pieter Van Gorp, and Daniel Varro. *Model-driven Software Development - Volume II of Research and Practice in Software Engineering*, chapter Using Graph Transformation for Practical Model Driven Software Engineering. Springer-Verlag, 2005.
48. Albert Zündorf. *Rigorous Object Oriented Software Development*. PhD thesis, University of Paderborn, 2001.
49. T. Fischer, J. Niere, L. Torunski, and A. Zündorf. Story diagrams: A new graph rewrite language based on the unified modeling language. In G. Engels and G. Rozenberg, editors, *Proc. of the* 6^{th} *International Workshop on Theory and Application of Graph Transformation (TAGT), Paderborn, Germany*, LNCS 1764. Springer Verlag, 1998.
50. Olaf Muliawan, Hans Schippers, and Pieter Van Gorp. Model driven, Template based, Model Transformer (MoTMoT). http://motmot.sourceforge.net/, 2005.
51. Remco M. Dijkman. Yatl4mdr: A model transformation engine and ocl checker for the netbeans meta-data repository.
52. Pieter Van Gorp, Hans Stenten, Tom Mens, and Serge Demeyer. Towards automating source-consistent UML refactorings. In *Proceedings of « UML » 2003 – The Unified Modeling Language*. Springer-Verlag, 2003.
53. Eric Cariou, Raphaël Marvie, Lionel Seinturier, and Laurence Duchien. OCL for the specification of model transformation contracts. In Octavian Patrascoiu, editor, *OCL and Model Driven Engineering, UML 2004 Conference Workshop, October 12, 2004, Lisbon, Portugal*, pages 69–83. University of Kent, 2004.
54. Tom Mens, Krzysztof Czarnecki, and Pieter Van Gorp. 04101 discussion – a taxonomy of model transformations [online]. In Jean Bezivin and Reiko Heckel, editors, *Language Engineering for Model-Driven Software Development*, number 04101 in Dagstuhl Seminar Proceedings. Internationales Begegnungs- und Forschungszentrum (IBFI), Schloss Dagstuhl, Germany, 2005. Extended version submitted to GraMoT'05.
55. David Akehurst, Stuart Kent, and Octavian Patrascoiu. A Relational Approach to Defining and Implementing Transformations in Metamodels. *Software and Systems Modeling*, 2(4):215–239, December 2003.
56. David H. Akehurst. *Model Translation: A UML-based specification technique and active implementation approach*. PhD thesis, December 2000. (unknown variable note).
57. M. W. A. Steen, D. H. Akehurst, H. W. L. ter Doest, and M. M. Lankhorst. Supporting viewpoint-oriented enterprise architecture. In *Enterprise Distributed Object Computing Conference, (EDOC)*, pages 201–211, 2004.
58. Robert Balzer. Tolerating inconsistency. In *ICSE '91: Proceedings of the 13th international conference on Software engineering*, pages 158–165, Los Alamitos, CA, USA, 1991. IEEE Computer Society Press.

59. Ben Moszkowski. *Executing temporal logic programs*. Cambridge University Press, New York, NY, USA, 1986.
60. Anthony Finkelstein, Dov M. Gabbay, Anthony Hunter, Jeff Kramer, and Bashar Nuseibeh. Inconsistency handling in multi-perspective specifications. In *ESEC '93: Proceedings of the 4th European Software Engineering Conference on Software Engineering*, pages 84–99, London, UK, 1993. Springer-Verlag.
61. Hartmunt Ehrig and Michael Pfender and Hans Jürgen Schneider. Graph-Grammars: An Algebraic Approach. In *FOCS*, pages 167–180, 1973.
62. Reiko Heckel and Annika Wagner. Ensuring consistency of conditional graph rewriting – a constructive approach. *Electronic Notes in Theorectical Computer Science*, 2, 1995. Joint COMPUGRAPH/SEMAGRAPH Workshop on Graph Rewriting and Computation.
63. Gabi Täntzer and Olga Runge. AGG – The Attributed Graph Grammar System: A Development Environment for Attributed Graph Transformation Systems. http://tfs.cs.tu-berlin.de/agg/, 2005.
64. Tom Mens. *A Formal Foundation for Object-Oriented Software Evolution*. PhD thesis, Department of Computer Science, Vrije Universiteit Brussel, 1999.
65. Tom Mens, Gabriele Taentzer, and Olga Runge. Detecting structural refactoring conflicts using critical pair analysis. *Electronic Notes in Theoretical Computer Science*, 127(3):113–128, April 2005.
66. D. Akehurst, H. Bowman, J. Bryans, and J. Derrick. A manual for a model checker for stochastic automata. Technical Report 5-00, University of Kent, Canterbury, 2000.
67. Roswitha Bardohl. A visual environment for visual languages. *Science of Computer Programming*, 44(2):181–203, August 2002.