Electronic Communications of the EASST
Volume 41 (2011)

Proceedings of the Tenth International Workshop on Graph Transformation and Visual Modeling Techniques (GTVMT 2011)

Automated Model Synchronization: A Case Study on UML with Maude

Artur Boronat, José Meseguer

14 pages

Guest Editors: Fabio Gadducci, Leonardo Mariani
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Automated Model Synchronization: A Case Study on UML with Maude

Artur Boronat[1], **José Meseguer**[2]

[1]aboronat@mcs.le.ac.uk, University of Leicester
[2]meseguer@uiuc.edu, University of Illinois at Urbana-Champaign

Abstract: Design specifications of software-intensive systems involve models that have been defined with different modelling languages for different purposes. Hence, a specification can be seen as the description of a system from multiple viewpoints, each providing domain-specific constructs for modelling the system in a more precise way. Such heterogeneity of models can jeopardize the consistency of the specification, because updates in one viewpoint may cause unpredictable design errors in other viewpoints, which can then be transferred to the implementation. OMG's Meta-Object Facility enhances the automation of the model consistency management by providing a uniform format for different modelling languages. In this paper, we illustrate a technique, based on rewriting logic and on strategies for finding inconsistencies in MOF-based heterogeneous specifications and for resolving them in an automated way.

Keywords: model synchronization, inconsistency repair plans, MOF, Maude.

1 Introduction

Software-intensive systems comprise a wide variety of heterogeneous software artefacts, including requirements, design models, program code, test suites, configuration files and documentation, which are usually developed by distributed teams. These artefacts describe parts of the same system from different points of view and at different levels of abstraction, hence overlapping in various ways. In this setting, evolution is a challenging, expensive task since changes can result in unanticipated inconsistencies and incompleteness [AC07].

To enhance the automated management of software evolution, model-driven software development methods advocate the use of models as first-class citizens to abstract relevant features of a system from implementation details. Within this context, OMG's Meta-Object Facility [OMG06] provides a common metadata infrastructure in which the abstract syntax of modelling languages is defined by means of so-called metamodels, which can be viewed as type graphs, so that software specifications can be represented as (typed attributed) instance graphs.

Two main approaches to manage inconsistencies in model-based heterogeneous specifications can be distinguished by considering how a set of consistent specifications is defined: either by means of coupled graph grammars, such as the case of triple-graph grammars [SK08], or by means of MOF metamodels with constraints. In the first case, consistency is characterized by membership in the resulting graph language, and automatically-generated transformations deal with incompleteness of views while consistency is ensured by construction. In the second case,

materialization of specification inconsistencies during the design process is assumed to be a natural, unpredictable (and even desirable) event [FGH+94], for which means are provided to identify inconsistencies and to fix them.

In the second approach, given a set of metamodels and a set of mappings among them defining inconsistency constraints, inconsistencies in a multi-domain specification have to be found and repaired until consistency is restored, i.e., until no inconsistency constraint is satisfied. Achieving a practical solution to this problem involves addressing several challenges, such as scalability and usability issues [SMBB10]: once a conflict is detected, an automated mechanism should find a number of resolution choices; resolution of conflicts should minimize the creation of new conflicts; the order in which inconsistencies are resolved is important to reproduce the changes to be applied, due to possible conflicts between updates. A sequence of updates that restores the consistency in a specification is called a *repair plan*.

In this work, we follow this second approach, focusing on a novel encoding[1], in Maude, of state-of-the-art techniques for defining and managing design inconsistencies so that they can be applied in MOF multi-domain specifications. In particular, inconsistency constraints are expressed as parametric model propositions that are formally defined in equational logic. Once inconsistencies are detected, resolution choices are locally obtained following the user-defined choice generator function technique proposed in [Egy07]. Our approach uses rewriting logic to define a search space of possible repair plans from a given inconsistent specification, and exploits Maude's pattern matching algorithm modulo associativity, commutativity and identity (ACU) and search capabilities to find an efficient repair plan that brings an inconsistent heterogeneous specification to a consistent state, where efficiency is estimated by means of the number of inconsistencies that need to be fixed after the application of an update. We use the Breadth-First Search (BFS) algorithm built within Maude's search command. We explain how several strategies to search consistent specifications can be expressed in a succinct way.

The paper is structured as follows: §2 summarises the formalization of models as graphs in rewriting logic and a dual representation of models in our approach; §3 details the main contribution of the paper by defining the chief constructs to identify inconsistencies, to provide inconsistency resolution choices and to find efficient repair plans by exploring the search space of inconsistent specifications using a BFS; §4 provides an account of related approaches; and §5 draws some final conclusions on the approach.

2 An Algebraic Approach to Encode and Manipulate MOF Models

2.1 Rewriting Logic

A theory in membership equational logic [Mes98] is used to define algebraic data types, specifying their types using sorts and subsorts, their operations using equationally-defined operators and structural properties using equations and memberships. Such a theory is defined as a pair $(\Sigma, E \cup A)$, where Σ is a signature specifying sorts, subsorts, kinds and operators in the theory, E is a collection of statements (equations and memberships, possibly conditional) and A

[1] Some familiarity with algebra and equational logic is assumed, mainly concerning to the notions of algebra, binary operations, and equational specifications of operations.

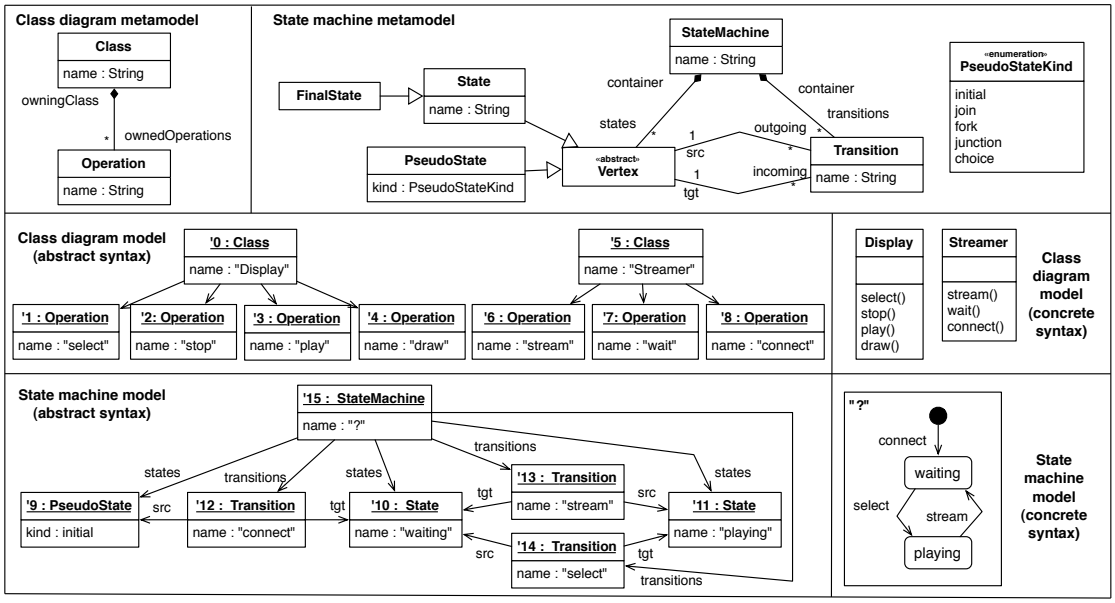

Figure 1: Simplified version of UML metamodel for class diagrams (top left) and for state machines (top right); syntactic representation of a class diagram (middle) and of a state machine (bottom), both in abstract and concrete syntax.

is a set of structural axioms, such as associativity (`assoc`), commutativity (`comm`) and identity (`id: element`), defined for some binary operators. Under the assumption that the equations E are confluent, terminating and sort-decreasing, the theory $(\Sigma, E \cup A)$ defines the initial model $T_{(\Sigma, E \cup A)}$, which is the quotient of the Σ−algebra T_Σ of ground terms modulo the equations and axioms $E \cup A$. In rewriting logic [BM06], a membership theory $(\Sigma, E \cup A)$ can be extended with rewrite rules R specifying local concurrent transitions in a system, whose states are algebraically characterized by the membership theory. In this work, we use membership theories for formalizing metamodels and for representing and manipulating models as directed graphs; and we use rewriting logic to analyze sequences of model manipulation events that enforce the consistency in a multi-domain specification in the presence of inconsistencies.

2.2 A State-based View of MOF Models

In this section we recall an algebraic representation of MOF models as (equivalence classes of) terms in an equational theory and explain how models are semantically treated as (typed attributed) graphs [BM10].

The abstract syntax of models is defined through metamodels. A metamodel is a collection of metaclasses that contain properties, which can be either attributes – typed with basic datatypes – or references – typed with other metaclasses. Metaclasses can specialize other metaclasses through metaclass inheritance relationships. In Fig. 1 (top), an excerpt of the UML metamodel for class diagrams and for state machines is shown.

A metamodel is syntactically formalized as an order-sorted signature Σ and a set of equations.

Σ can be split into a generic part providing constructs (sorts, subsorts, and operators) for defining directed graphs of objects, and a metamodel-specific part containing type-relative information (metaclass names and properties). In the generic part, an object in a model is represented by a triple `< O : C | PS >` using the operator

```
op <_:_|_> :  Oid Cid PropertySet -> Object .
```

so that `O` is a unique object identifier[2], `C` is a class name, and `PS` is a record where each field represents either a slot (an attribute value) or a reference (a collection of pointers to other objects). Objects can be grouped in collections of objects by means of an associative, commutative union operator (denoted by juxtaposition) `__` with identity `none`:

```
subsort Object < ObjCol .  op none :  -> ObjCol .
op __ :  ObjCol ObjCol -> ObjCol [assoc comm id:  none] .
```

In the metamodel-specific part of the signature Σ, there is a sort for each metaclass name and there is a constant defining this sort if the metaclass is not abstract. Metaclass inheritance is formalized by means of subsort relationships between the sorts corresponding to each metaclass. Taking into account the resulting subsort ordering, maximal sorts in this ordering are declared as subsorts of the sort `Cid`. In addition, properties defined for each metaclass are encoded as constructors[3] of the sort `Property` defining fields for the record `PropertySet`. The classes Vertex, State and PseudoState from the metamodel in Fig. 1 are encoded as follows:

```
sorts sm/State sm/PseudoState sm/Vertex .
subsorts sm/State sm/PseudoState < sm/Vertex .
op sm/Vertex/outgoing`:_ :  OidSet -> Property .
op sm/Vertex/incoming`:_ :  OidSet -> Property .
op sm/State :  -> sm/State .
op sm/State/name`:_ :  String -> Property .
op sm/PseudoState/kind`:_ :  PseudoStateEnum -> Property .
```

In this work, the carrier of the sort `ObjCol`[4] in the initial algebra $T_{\Sigma, E \cup A}$, associated with the equational theory $(\Sigma, E \cup A)$ corresponding to a metamodel, provides the language of models that conform to this metamodel, i.e., its algebraic semantics [BM10]. Note that each such model corresponds to an instance graph whose nodes are associated with metaclass names defined in the metamodel. The abstract syntax graph of an excerpt of the UML class diagram in Fig. 1 (middle left) is represented as an (ACU) term as follows:

```
< '5 :  Class | name :  "Streamer", ownedOperations :  '6 '7 '8 >
< '6 :  Operation | name :  "stream", owningClass :  '5 >
< '7 :  Operation | name :  "wait", owningClass :  '5 >
< '8 :  Operation | name :  "connect", owningClass :  '5 >
```

2.3 An Event-based View of MOF Models

In this section we present an algebra inspired in the MOF reflective API, initially presented in [Bor07] and applied in [BHM09], for creating and manipulating MOF models in heterogeneous specifications, i.e., specifications that consist of models conforming to different metamodels.

[2] We define identifiers using literals prefixed with a quote, such as `'a` or `'1`, using the built-in datatype `Qid`, which is defined as a subsort of the sort `Oid`.

[3] We use the qualified name of properties to avoid name clashes when several metamodels are taken into account. However we will abbreviate their name to their last suffix when there is no room for confusion.

[4] In previous work, we used the sort `Model` to define the language of well-formed models for a given metamodel. In this paper, we simplify the sort structure to provide a simpler presentation of models as terms.

We define the representation of a multi-domain specification, the notion of model event and how events can act on a specification.

Domains. A domain encapsulates a specific model within a heterogeneous specification and tags it with a label name that identifies the domain and with the name of the metamodel, indicating the type of the domain. Syntactically, a domain is given as a structured object that has a unique name, a type name corresponding to the name of the metamodel and a single property, for which we simply omit the name, containing a collection of objects representing the model:

```
sorts Domain Multimodel .
op <_:_|_> :  Oid Cid ObjCol -> Domain .
```

A multi-domain specification is given as a multiset of domains where the uniqueness of the domain names is assumed implicitly.

```
subsort Domain < Multimodel .
op none :  -> Multimodel .
op __ :  Multimodel Multimodel -> Multimodel [comm assoc id:  none] .
```

Model events. A model event syntactically represents an atomic change in a model, mainly concerning the creation and destruction of objects, and the manipulation of their properties. Each possible change is represented by an operator that always carries as argument the identifier of the object that is manipulated. Assuming that the name space of object identifiers is global, i.e., object identifiers are unique in a multi-domain specification, `create(D,C,O)` represents the creation of an object with identifier `O` of type `C` in the model of the domain `D`; `destroy(O)` represents the removal of the object with identifier `O`; `set(O, P, V)` represents the initialization of the attribute with name `P` with the value `V` in the object with identifier `O`; and `set(O1,P,O2)` represents the addition of the identifier `O2` to the collection of identifiers in the value of the reference with name `P` in the object with identifier `O1`.

```
sort ModelEvent .  op create :  Cid Oid -> ModelEvent .
op destroy :  Oid -> ModelEvent .
op set :  Oid PropName String -> ModelEvent .
op set :  Oid PropName OidSet -> ModelEvent .
```

A complex change can be represented as a sequence of model events. This is a achieved through the associative concatenation operator `_;_`:

```
sort ModelEventSeq .  subsort ModelEvent < ModelEventSeq .
op _;_ :  ModelEventSeq ModelEventSeq -> ModelEventSeq [assoc] .
```

A model can then be created or modified by a sequence of model events. The sequence of events that creates the class `"Display"` with the operation `"select"` in the class diagram given in Fig. 1 corresponds to the following sequence `create(cd, Class, '0) ; set('0, name, "Display") ; create(cd, Operation, '1) ; set('1, name, "select") ; set('1, owningClass, '0) ; set('0, ownedOperations, '1).`

Converting from event-based to state-based representation. In our approach the canonical representation of a model is given by its state-based representation. In fact, the state-based representation of a model corresponds to the representative of an equivalence class of model event sequences, those that lead to the same canonical form (up to name isomorphism with respect to object identifiers). The following operator enables the application of events to a term representing the state-based view of the model.

```
op [_]_ :  [Multimodel] [ModelEventSeq] -> [Multimodel] .
```

The semantics of this operator is equationally defined below and corresponds to the intuition described for each model event above. Note that the operator is only partially defined[5].

```
ceq [ < DN : MN | OC > M ] create(DN, C, O)
= < DN : MN | new( C, O ) OC > M if not (O in OC) .
ceq [ < DN : MN | < O : C | PS > OC > M ] destroy(O)
= < DN : MN | OC > M if nac("noDanglingEdge", oid :  O, < DN : MN | OC > M) .
eq [ < DN : MN | < O : C | PN : V1, PS > OC > M ] set(O, PN, V2)
= < DN : MN | < O : C | PN : V2, PS > OC > M .
ceq [ < DN : MN | < O1 :  C | PN : OS, PS > OC > M ] set(O1, PN, O2)
= < DN : MN | < O1 :  C | PN : O2 OS, PS > OC > M if O2 in OC .
eq [ < DN : MN | < O1 :  C | PN : OS, PS > OC > M ] set(O1, PN, null)
= < DN : MN | < O1 :  C | PN : null, PS > OC > M .
```

where `new(C,O)` is an operator defined with an equation for each metaclass and is metamodel dependent. An object can only be destroyed if it does not leave dangling edges. This is checked by the negative application condition `"noDanglingEdge"`, which checks that an object with a pointer `O` does not exist. The `nac` function enables the application of a rule that calls it when a specific pattern is *not* found in the multi-domain specification. Attribute values can be unset with the event `set( O, PN, V2 )` by using the default value of the attribute as new value `V2`, while references can be unset by using the value `null` in the event `set( O, PN, null )`.

3 An Approach for Detecting and Resolving Model Inconsistencies

In this section, the core components of our approach are introduced and are illustrated with a UML case study, adapted from [ELF08]: definition and detection of inconsistencies; generation of repair event sequences for a specific inconsistency; and generation of repair plans.

3.1 Model Inconsistency Relation

In our approach, a heterogeneous MOF specification is syntactically encoded as a multiset of domains containing viewpoint models conforming to different metamodels. Nonetheless, since these models are regarded as views of the same system, there is usually an overlap among them, so that the same information may be represented from different points of view and with different modelling languages. This overlap is defined by means of syntactic or structural constraints and by means of semantic constraints. Since the goal in this work is to find inconsistencies, we encode the negation of such consistency constraints by means of an inconsistency satisfaction relation $m \vDash i(o,p)$, where the existence of inconsistencies in the multi-domain specification m is determined by the truth value of a proposition i parameterized with an object identifier o and a property name p. In the example, two model predicates are defined in a similar way as in [BHM09] for indicating violations of the requirements that a state machine must correspond to a class in the class diagram (`classNotDefined`), and that transitions in a state machine must be labelled with a method of the class associated with the state machine (`methodNotDefined`). The satisfaction operator for the inconsistency relation is as follows:

[5] The use of brackets indicates that the operator is not defined for the entire carrier of the sort `Multimodel`, i.e., it is partial and is therefore defined over its kind.

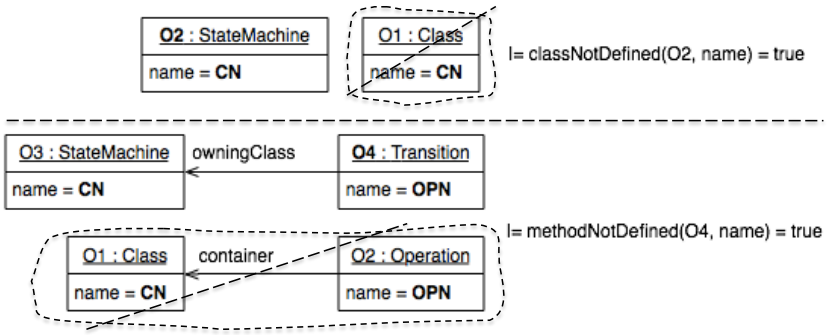

Figure 2: Graphical representation of the inconsistency relation of the case study.

```
op _|=_ : Multimodel InconsistencyProp -> Bool [frozen] .
```

where the attribute `frozen` avoids the application of rewrite rules to the multimodel.

The inconsistency constraints defined in Fig. 2 are expressed as equations defining the inconsistency satisfaction relation as follows:

```
op classNotDefined : Oid PropName -> InconsistencyProp .
ceq M |= classNotDefined( O2, sm/StateMachine/name ) = true
if < DN : MN | < O2 : StateMachine | name : CN, PS2 > OC2 > M2 := M
 /\ nac("cnd-nac", ClassName : CN, M) = true .
op methodNotDefined : Oid PropName -> InconsistencyProp .
ceq M |= methodNotDefined( O4, sm/Transition/name ) = true
if < DN : MN | < O3 : StateMachine | name : CN, PS3 >
 < O4 : Transition | name : OPN, container : O3, PS4 > OC2 > M2 := M
 /\ nac("mnd-nac", ClassName : CN OperationName : OPN, M) = true
```

where `nac("cnd-nac", ClassName : CN, M)` checks whether there is not a class with name `CN` in the domain `cd` in the multi-domain specification `M`, and `nac("mnd-nac", ClassName : CN OperationName : OPN, M)` checks whether there is not a method with name `OPN` in the class with name `CN` in the domain `cd` in the multi-domain specification `M`. Finally, an equation that is applied in case an inconsistency cannot be found is added for completing the equational specification of the inconsistency predicate:

```
eq M:Multimodel |= IP:InconsistencyProp = false [owise] .
```

The purpose of the inconsistency relation is twofold: the definition of inconsistencies and their detection. When a given inconsistency proposition is satisfied, the parameters of the proposition identify the location of the inconsistency as indicated in the following section.

3.2 Inconsistency Location

Inconsistencies occur when certain model elements in one or several domain models satisfy the inconsistency relation with regard to a given inconsistency proposition, hence violating the associated consistency constraint. Most inconsistencies can be avoided by manipulating property values (including both attributes and references) in the conflictive objects of a domain model. Hence, in our approach an inconsistency is uniquely determined by a pair $< o, p >$, formed by an object identifier `o` and a (qualified) property name `p` in the object identified by `o`.

```
sorts Inconsistency .
op <_`,_> :  Oid PropName -> Inconsistency .
```

Sets of inconsistencies are defined in the usual way through a binary operation that is associative, commutative and idempotent, syntactically represented with the juxtaposition operator __. In the multi-domain specification in Fig. 1, two inconsistencies are found with regard to the inconsistency relation in Fig. 2, namely `<'14, sm/Transition/name>` and `<'15, sm/StateMachine/name>`.

3.3 Choice Generator Function

Once inconsistencies are detected, an automated approach for conflict resolution must provide one or more possible solutions in order to fix the inconsistency. Given that the location of an inconsistency is determined by a property in a specific object, resolution choices are syntactically constrained by the type of this location. Hence, they are specific to one metamodel. We follow the proposal from [ELF08], whereby resolution choices are manually encoded in choice generator functions independently of inconsistency constraints. In addition, choices are provided as finite sets of elements that are extracted from the multi-domain specification using a query mechanism, imposing a reasonable bound on the number of decisions that have to be explored. This reduces the amount of resolution decisions that have to be encoded from `#inconsistency constraints * #location types` to `#location types` only, enhancing maintenability. Choices for a particular inconsistency location type are given by the following function:

```
op repairChoices :  Multimodel Oid PropName ModelEventSeq -> ModelEventSeq .
```

which has to be refined for each inconsistency type location. This function takes as arguments a multi-domain specification and the location of an inconsistency (identified by an object identifier and a property name), and generates a set[6] of repair actions, where a repair action is a sequence of events that eliminates the inconsistency. In the example, possible choices for fixing the name of the state machine are the set of names of classes in the class diagram. The following equation, generates a `set` event with the name of each class in the class diagram:

```
eq  repairChoices(< DN : cd | < O1 :  Class | name :  CN, PS > OC > M,
    O, sm/StateMachine/name, RAS )
=   repairChoices( < DN : cd | OC > M,
    O, sm/StateMachine/name, (set(O, sm/StateMachine/name, CN) RAS) ) .
```

Similarly, possible solutions for fixing the label of a transition, when it does not correspond to a method in the class associated with the state machine, are the names of the methods of the corresponding class.

[6] This set is also passed as argument so that it can be recursively built through the application of equations.

```
eq  repairChoices(
    < DN1 :  cd | < O1 :  Class | name :  CN, ownedOperations :  O2 OS1, PS1 >
    < O2 :  Operation | name :  OPN, PS2 > OC1 >
    < DN2 :  sm |
    < O3 :  StateMachine | name :  CN, transitions :  O OS2, PS3 > OC2 > M,
    O, sm/Transition/name, RAS )
=   repairChoices(
    < DN1 :  cd | < O1 :  Class | name :  CN, ownedOperations :  OS1, PS1 > OC1 >
    < DN2 :  sm |
    < O3 :  StateMachine | name :  CN, transitions :  O OS2, PS3 > OC2 > M,
    O, sm/Transition/name, (set(O, sm/Transition/name, OPN) RAS) ) .
```

Note that the second equation only returns choices if the name of the class associated with the state machine belongs to an existing class. Hence, in case both a class name and a method name are found to be inconsistent, `repairChoices` will only provide choices for fixing the method name after the class name has been fixed. The following equation returns the set of choices produced for a specific inconsistency when no more repair actions can be generated:

```
eq repairChoices( M, O, PN, RPS ) = RPS [owise] .
```

3.4 Repair Plan Selection and Generation

In the previous section, the approach for detecting inconsistencies and for proposing resolution choices has been explained. In this section, a mechanism for building repair plans as sequences of model events that remove inconsistencies from the specification is developed.

Definition 1 (Repair action) Given a multi-domain specification m and an inconsistency $< o, p >$ such that $m \models i(o,p)$ for a specific inconsistency proposition i, a repair action ra for this inconsistency, i.e., $ra \in repairChoices(m,o,p,\varnothing)$ is a sequence of model events such that $[m]\ ra \not\models i(o,p)$.

Since the inconsistency satisfaction relation $\models$ can be defined for different inconsistency propositions and a multi-domain specification may contain several inconsistencies of different types, we extend the notion of repair action to the notion of repair plan as follows:

Definition 2 (Repair plan) Given a multi-domain specification m, a set $\mathcal{I}$ of inconsistency propositions, and an inconsistency satisfaction relation $\models$, a repair plan for an inconsistent specification m is a sequence $rp = ra_1;\ldots;ra_n$ such that $ra_i \in repairChoices(m,o,p,\varnothing)$ for an inconsistency $< o,p >$ in m, and $[m]\ rp \not\models I$ for all inconsistency propositions I in $\mathcal{I}$.

Note that repairing a specific inconsistency may result in more inconsistencies. This definition captures the idea that a repair plan is a sequence of events that leads to a consistent model.

The exploration of possible repair plans is achieved by specifying a conditional rule that, given a specific inconsistency, chooses a repair action from the set of choices provided by the function `repairChoices` for the inconsistency. The notion of repair plan is useful because it enables the analysis of sequences of repair actions corresponding to different inconsistencies and their composition, instead of considering the resolution of one inconsistency at a time.

In our analysis technique, the notion of state in the search space involves the dual view of

a model as a pair `[ M | ES ]`, where `M` is a multi-domain specification where each domain includes the state-based representation of a viewpoint model, and `ES` is a model event sequence providing the event-based representation of the same multi-domain specification. The search space is generated by applying a single rule with two sources of non-determinism: *(1)* several inconsistencies can occur in a multi-domain specification; and *(2)* for each inconsistency there may be several repair choices. We consider three ways of analysing choices for resolving a specific inconsistency: *(i)* by brute force, which allows us to explore all choices for a specific inconsistency; *(ii)* by an efficient repair-driven strategy, which constrains the number of repairing choices that can be applied for a given inconsistency; and *(iii)* by extending the second strategy to all inconsistencies.

Brute force. The following conditional rule obtains the set of inconsistencies through the function `getIS`, which recursively collects a set of inconsistencies in its second argument, and chooses one, namely `<O,PN>`, modulo associativity commutativity in the first matching equation in the condition of the rule. The choices for repairing this inconsistency are retrieved by means of the function `repairChoices`, from which one repair action is chosen, namely `RA`, which is applied to the multi-domain specification in the right hand side of the equation in order to fix the selected inconsistency. We also append this repair action to the current repair plan in the second component of the state. The rule is shown in Maude format as follows:

```
crl  [ M | RP ] => [ [ M ] RA | RP ; RA ]
if   < O , PN > IS := getIS(M, none)
     /\ RA RAS := repairChoices(M, O, PN, noAction) .
```

Efficient repair-driven strategy. The rule above is applied with a brute force strategy, in which all choices for fixing inconsistencies are explored, even those that insert more inconsistencies or that would not make sense from the user point of view. For example, if the name of the class associated with the state machine happens to be inconsistent with the class diagram, the choices that would be proposed by `repairChoices` are: `"Streamer"` and `"Display"`. However, choosing `"Display"` would make most of the labels in the state machine inconsistent, whereas choosing `"Streamer"` would lead to a model with only one inconsistency left.

Definition 3 (Extended repair action) Given a multi-domain specification m and an inconsistency relation $\models$, an extended repair action is a pair (ra,n), where ra is a repair action, and n is a natural number indicating the number of inconsistencies w.r.t. $\models$ in the specification $[m]\ ra$, i.e., the number of inconsistencies left in the model m after applying the repair action ra.

Definition 4 (Strict order over repair actions) Given a multi-domain specification m and an inconsistency relation $\models$, we define the strict order $(A,<)_{m \models <o,p>}$ over the set A of extended repair actions (ra,n) for a given inconsistency $<o,p>$, where $ra \in repairChoices(m,o,p,\varnothing)$, by extending the strict ordering over the component with the natural number as follows:

$$(ra,n) < (ra',n') \quad if \ \ n < n'$$

We refine the rule above with a sensible heuristic to select only minimal extended actions with regard to this strict ordering, that is, we select repair actions that lead to specifications

with a minimal number of inconsistencies. The refinement consists in the addition of one more matching equation in the condition of the rule. This equation computes the set of extended repair actions by means of the function `countIS` and selects the repair actions from the set of minimal extended repair actions through the function `selectOptimalRAS` as explained above[7].

```
crl  [ M | RP ] => [ [ M ] RA | RP ; RA ]
if   < O , PN > IS := getIS(M, none)
     /\ RAS := repairChoices(M, O, PN, noAction)
     /\ RA RAS2 := selectOptimalRAS(countIS( M, RAS, empty ), MAX, RAS) .
```

A variant of this strategy is defined by choosing a repair action from the set of minimal extended repair actions that corresponds to all inconsistencies, thus reducing the amount of choices due to different inconsistencies:

```
crl  [ M | RP ] => [ [ M ] RA | RP ; RA ]
if   < O , PN > IS := getIS(M, none)
     /\ RAS := repairAllChoices(M, O, PN, noAction)
     /\ RA RAS2 := selectOptimalRAS(countIS( M, RAS, empty ), MAX, RAS) .
```

where `repairAllChoices` is the extension of the function `repairChoices` to all inconsistencies.

Repair Plan Generation. To generate repair plans for a given inconsistent multi-domain specification, we can use Maude's `search` command in order to explore the search space generated by each one of the aforementioned rules using Breadth-First Search (BFS). The use of BFS ensures that a repair plan will be found if it exists.

As initial state, we use the pair `[ m | noEvent ]`, where `m` is an equationally-defined constant representing the inconsistent specification and `noEvent` represents the empty repair plan. We can use the search command to obtain a pair matching the pattern `[M:Multimodel|RA:RepairPlan]`, where `M:Multimodel` will be assigned to a consistent specification and `RA:RepairPlan` will be assigned to the repair action that can be used to obtain the consistent specification in `M` from `m`. This can be achieved by using the search option `=>!` to obtain a canonical form of the pair instantiating `[M:Multimodel|RA:RepairPlan]` such that the condition `getIS(M:Multimodel, none) == none` is satisfied, i.e., no more inconsistencies are left. We can explore one solution through the option `[1]` as follows:

```
search [1] [ m | noEvent ] =>!  [ M:Multimodel | RA:RepairPlan ]
such that getIS(M:Multimodel, none) == none .
```

obtaining the first solution:

```
Solution 1 (state 8)
states: 26  rewrites: 2324 in 18ms cpu (18ms real) (126558 rewrites/second)
M:Multimodel --> ...
RA:RepairPlan --> set('15, sm/StateMachine/name, "Streamer") ;
set('12, sm/Transition/name, "connect") ;
set('13, sm/Transition/name, "connect")
```

To obtain the next solution we can use the command `cont 1 .`, or just `cont .` to resume all remaining solutions. Alternatively, we can execute the search command without the option `[1]` to list all solutions directly.

[7] The complete specification of these functions can be found in the appendix.

4 Related work

In [ELF08], the authors provide an scalable mechanism for evaluating choices for fixing inconsistencies in UML design models at runtime. This mechanism draws on the generator functions provided in [Egy07] for obtaining repair choices for a given inconsistency. In addition, it filters out those repair choices that create other inconsistencies by checking that the elements to be modified do not violate other inconsistency rules in an incremental way, that is, only inconsistency rules that apply to these model elements are checked. In this approach, valid repair choices are shown to the user who must choose one. Hence, only one conflict can be manually resolved at a time resulting in a fine-grained semi-automated resolution approach.

An approach for generating complex repair plans for resolving model inconsistencies in an automated way is presented in [SMBB10]. This approach is based on an operation-based representation of models [BMMM08] where inconsistency rules are encoded as cause detection rules using Prolog as underlying formalism. These rules detect which actions caused inconsistencies instead of identifying which elements are inconsistent. The Iterative Deepening Depth-first Search Strategy (IDDFSS) is used to find the best repair plan that leaves the model in a consistent state. Some heuristics are used to guide the search by assuming that the last changes in the model are the most likely to be inconsistent. In our approach, we use both a state-based representation of models to find specific inconsistent model elements and a similar operation-based representation for generating repair plans and for checking their validity. This dual view of a model comes at a low computational cost of conversion, since updates are determined by a specific model element. In this way, we can use the state-based view of a model, usually available in MOF-compliant modelling repositories, or the operation-based view when the order of actions provided by the user is relevant. We have shown how to use Maude's search BFS algorithm for finding efficient repair plans to obtain consistent models. A key difference with the approach above is that the state-based view of a model corresponds to an equivalence class of event sequences, namely those that lead to the same model up to renaming, hence reducing the number of states that need to be explored in the search space. A more comprehensive comparison with this approach is left for future work due to the current lack of an incremental consistency checker.

The approach in [SMSJ03] uses description logic to detect and resolve the inconsistencies among UML class diagrams, sequence diagrams and state machines that result from the evolution of such models. The underlying formalism allows the authors to dismiss the closed-world assumption to enable the definition of incomplete models; whilst in our work inconsistencies are checked over a *closed world*, the multi-domain specification, and the lack of inconsistencies is understood as a consistent specification. Our assumption seems reasonable in common modelling scenarios and tools, where inconsistencies are checked in multi-domain specifications that may have been developed by different teams but that are available as a whole nonetheless. In [SMSJ03], strategies for providing repair plans were not studied.

Graph-transformation theory is a well-established formalism for defining visual languages, allowing for the analysis of consistent model-based specifications. In particular, triple-graph grammars (TGGs) [SK08] are used to define consistency relations between graph grammars, so that forward and backward (non-deleting) translators can be automatically generated. To guarantee that generated translators effectively compute synchronisation policies, properties to determine their functional nature were studied in [EEHP09, HEOG10]. An analysis of bidirectional

model transformations based on TGGs with respect to information preservation is presented in [EEE+07]. A recent approach [LMT11] considers TGGs to relate models with different syntax by considering different metamodels and triple graph constraints.

Maude has been previously used for encoding UML consistency constraints in a QVT Relations dialect and for resolving inconsistencies using rewrite rules [LMÁ09], pursuing the use of MDA standards and the development of a CASE tool. The main aim in our approach has been the design of efficient mechanisms for exploring and resolving inconsistencies using inconsistency constraints, local definitions of repair actions and search strategies.

5 Conclusions

In this paper, we have presented search strategies for identifying model inconsistencies in heterogeneous MOF-specifications and for automating inconsistency resolution using Maude, and have illustrated them with a UML case study. In particular, Maude's search command is used to find consistent models using a BFS strategy and two uniform-cost search strategies are used to improve the outcome of the initial strategy. In an extended version of this paper[8], the use of Maude's LTL model checker for finding model repair plans and a quantitative evaluation of the different search strategies are also discussed. Our technique for analyzing and repairing inconsistencies addresses the following software quality features: *maintenance*, our technique is generic and can be instantiated with a specific set of metamodels; *scalability*, the function `repairChoices` generates a finite set of repair actions by querying the specification, imposing a bound on the branching factor of the search tree for resolving a given inconsistency; *usability*, the syntactic representation of MOF models is isomorphic to the one used in the tool MOMENT2 [BHM09], hence an integration of our technique into EMF has already been proved feasible.

Acknowledgements: We are grateful to anonymous reviewers for their helpful comments. The first author has been supported by the University of Leicester. The second author has been supported by NSF Grant CCF 09-05584.

Bibliography

[AC07] M. Antkiewicz, K. Czarnecki. Design Space of Heterogeneous Synchronization. In *GTTSE*. LNCS 5235, pp. 3–46. 2007.

[BHM09] A. Boronat, R. Heckel, J. Meseguer. Rewriting Logic Semantics and Verification of Model Transformations. In *FASE*. Pp. 18–33. LNCS 5503, 2009.

[BM06] R. Bruni, J. Meseguer. Semantic foundations for generalized rewrite theories. *Theor. Comput. Sci.* 360(1-3):386–414, 2006.

[BM10] A. Boronat, J. Meseguer. An Algebraic Semantics for MOF. *Formal Aspects of Computing* 22:269–296, 2010.

[8] http://www.cs.le.ac.uk/~aboronat/papers/modelsynch11.pdf

[BMMM08] X. Blanc, I. Mounier, A. Mougenot, T. Mens. Detecting model inconsistency through operation-based model construction. In *ICSE*. Pp. 511–520. ACM, 2008.

[Bor07] A. Boronat. *MOMENT: a formal framework for MOdel manageMENT*. PhD thesis, Universitat Politènica de València (UPV), Spain, 2007.

[EEE+07] H. Ehrig, K. Ehrig, C. Ermel, F. Hermann, G. Taentzer. Information Preserving Bidirectional Model Transformations. In *FASE*. Pp. 72–86. LNCS 4422, 2007.

[EEHP09] H. Ehrig, C. Ermel, F. Hermann, U. Prange. On-the-Fly Construction, Correctness and Completeness of Model Transformations Based on Triple Graph Grammars. In *MODELS*. Pp. 241–255. LNCS 5795, 2009.

[Egy07] A. Egyed. Fixing Inconsistencies in UML Design Models. In *ICSE*. Pp. 292–301. IEEE Computer Society, 2007.

[ELF08] A. Egyed, E. Letier, A. Finkelstein. Generating and Evaluating Choices for Fixing Inconsistencies in UML Design Models. In *ASE '08*. Pp. 99–108. IEEE Computer Society, 2008.

[FGH+94] A. Finkelstein, D. Gabbay, A. Hunter, J. Kramer, B. Nuseibeh. Inconsistency handling in multiperspective specifications. *Software Engineering, IEEE Transactions on* 20(8):569–578, Aug. 1994.

[HEOG10] F. Hermann, H. Ehrig, F. Orejas, U. Golas. Formal Analysis of Functional Behaviour for Model Transformations based on Triple Graph Grammars. In *ICGT*. LNCS, 2010.

[LMÁ09] F. J. Lucas, F. Molina, J. A. T. Álvarez. A systematic review of UML model consistency management. *Information & Software Technology* 51(12):1631–1645, 2009.

[LMT11] Y. Lamo, F. Mantz, G. Taentzer. On the Relation of Meta-Modeling and Typed Graphs. In *GT-VMT'11*. ECEASST, 2011.

[Mes98] J. Meseguer. Membership Algebra as a Logical Framework for Equational Specification. In *WADT'97*. Pp. 18–61. LNCS 1376, 1998.

[OMG06] OMG. Meta Object Facility (MOF) 2.0 Core Specification (ptc/06-01-01). 2006.

[SK08] A. Schürr, F. Klar. 15 Years of Triple Graph Grammars. In *ICGT*. Pp. 411–425. LNCS 5214, 2008.

[SMBB10] M. A. A. da Silva, A. Mougenot, X. Blanc, R. Bendraou. Towards Automated Inconsistency Handling in Design Models. In *CAiSE*. LNCS 6051, pp. 348–362. Springer, 2010.

[SMSJ03] R. V. D. Straeten, T. Mens, J. Simmonds, V. Jonckers. Using Description Logic to Maintain Consistency between UML Models. In *UML*. LNCS 2863, pp. 326–340. Springer, 2003.