Towards a Unifying Conceptual Framework for Inconsistency Management Approaches: Definitions and Instantiations

Hubaux, Arnaud; Cleve, Anthony; Schobbens, Pierre-Yves; Keller, Anne; Muliawan, Olaf; Castro, Sergio; Mens, Kim; Deridder, Dirk; Van Der Straeten, Ragnhild

Publication date:
2009

PReCISE – FUNDP
University of Namur
Rue Grandgagnage, 21
B-5000 Namur
Belgium

TECHNICAL REPORT May 14, 2009

AUTHORS A. Hubaux, A. Cleve, P.-Y. Schobbens, A. Keller, O. Muliawan, S. Castro, K. Mens, D. Deridder, R. Van Der Straeten
APPROVED BY P. Heymans
EMAILS {ahu|acl|pys}@info.fundp.ac.be,
{anne.keller|olaf.muliawan}@ua.ac.be,
{sergio.castro|kim.mens}@uclouvain.be,
{dderidde|rvdstrae}@vub.ac.be
STATUS Draft version
REFERENCE P-CS-TR WP4CM-000001
PROJECT MoVES
FUNDING Interuniversity Attraction Poles Programme of the Belgian State of Belgian Science Policy

Towards a Unifying Conceptual Framework for Inconsistency Management Approaches

Definitions and Instantiations

THE PRESENT DOCUMENT IS AN EXTENDED VERSION OF A PAPER SUBMITTED TO MODELS 2009. THE TECHNICAL PART HAS OTHERWISE NOT BEEN PUBLISHED OR SUBMITTED ELSEWHERE.

Towards a Unifying Conceptual Framework for Inconsistency Management Approaches

Definitions and Instantiations

Arnaud Hubaux[1], Anthony Cleve[1], Pierre-Yves Schobbens[1], Anne Keller[2], Olaf Muliawan[2], Sergio Castro[3], Kim Mens[3], Dirk Deridder[4], and Ragnhild Van Der Straeten[4]

[1] PReCISE Research Centre, University of Namur
Rue Grandgagnage 21, 5000 Namur, Belgium
{ahu|acl|pys}@info.fundp.ac.be
[2] Department of Mathematics and Computer Science, Universiteit Antwerp
Middelheimlaan 1, 2020 Antwerpen, Belgium
{anne.keller|olaf.muliawan}@ua.ac.be
[3] Department of Computing Science and Engineering, Université catholique de Louvain
Place Ste Barbe 2, 1348 Louvain-la-Neuve, Belgium
{sergio.castro|kim.mens}@uclouvain.be
[4] Systems and Software Engineering Lab, Vrije Universiteit Brussel
Pleinlaan 2, 1050 Brussel, Belgium
{dderidde|rvdstrae}@vub.ac.be

Abstract. The problem of managing inconsistencies within and between models is omnipresent in software engineering. Over the years many different inconsistency management approaches have been proposed by the research community. Because of their large diversity of backgrounds and the diversity of models being considered, it is difficult to pinpoint what these approaches have in common and what not. As a result, researchers encounter difficulties when positioning and comparing their work with existing state-of-the-art, or when collaborating on or combining different approaches. Also, end-users have a hard time making informed decisions to select the most appropriate approach. To address these problems, we present a unifying conceptual framework of definitions and terminology, independent of any concrete inconsistency management approach or (modelling) language. The contribution is a formal framework providing a common understanding of what (in)consistency means, what inconsistency management involves and what assumptions are commonly made by existing approaches. The formalisation is also illustrated with four instantiations taken from different research fields.

Keywords. Inconsistency management, models, conceptual framework, formalism.

1 Introduction

The problem of managing inconsistencies is omnipresent in software engineering [1]. It appears at every phase of the software life-cycle, ranging from requirements, analysis, architecture, design, implementation and testing to maintenance and evolution.

Inconsistencies occur within and between models that are built with different languages and are of various types like requirements models, feature models, use cases, UML design models and even program code. Inconsistencies result from the violations of consistency conditions that can be as diverse as architectural and design guidelines, programming conventions, well-formedness rules and tests.

In spite of the importance of using models in software development and the need for dealing with inconsistencies, there is a lack of a unifying conceptual framework for model inconsistencies that encompasses all these different phases and models, and that allows to focus on what all these approaches have in common, and what distinguishes them. We encountered this particular problem in our MoVES research project (`moves.vub.ac.be`) where the lack of such common understanding hindered our comparison of different inconsistency management approaches. The MoVES partners belong to different communities among which requirements engineering, software product-line engineering, model-driven engineering, software maintenance and evolution and database engineering. Existing inconsistency classifications [2–7] focus on defining and classifying different types of inconsistencies that can be encountered. In contrast, the achievement of a common and unequivocal frame of reference and the specification of a basic set of requirements for inconsistency management approaches are barely tackled.

More specifically, one misses a reference framework for inconsistency management providing a common understanding of (1) what the different involved models and languages are, (2) what an inconsistency really means and (3) what inconsistency management tasks are supported. The contribution of this paper is to formally define such a framework that lays down the foundation for our future work where the proposed formalism will be used to: understand existing inconsistency management approaches, compare such approaches systematically, classify them, identify synergies between approaches and reuse approaches from one domain to another.

The framework proposed in this paper results from an incremental and bottom-up process exploiting the research background of the authors. Our starting point consisted of a collection of domain-specific approaches dedicated to the management of inconsistencies occuring within and between (1) UML models [8], (2) source code and structural regularities on that code [9], (3) database schemas and associated queries [10] and (4) model transformations [11]. The concrete instantiations of the framework in these areas we present bring forth early evidence of the applicability of the framework as a backbone for the unification of inconsistency management approaches.

The remainder of the paper is structured as follows. Section 2 introduces the unifying conceptual framework. After an intuitive description of our framework (2.1), we introduce an instantiation (2.2) that serves as an illustration for its rigourous definition (2.3). Section 3 details four instantiations of the formal definition of the framework. Section 4 explores related work. Section 5 proposes avenues for future work and concludes the paper.

2 Inconsistency Framework

In this section we introduce our unifying conceptual framework for inconsistency management approaches. We first introduce the involved concepts informally, then illustrate

them on a concrete inconsistency management approach, before finally providing some more formal definitions.

2.1 Intuitive definition

Following [1], *inconsistency management* minimally involves (1) the specification of consistency conditions to be checked, (2) the detection of violations of these conditions (called inconsistencies), and (3) the handling of these detected inconsistencies.

Specifying the consistency conditions boils down to defining the model(s) of interest and the conditions these models and their elements need to adhere to in order to be consistent. An inconsistency is *detected* whenever a particular consistency condition about the models and elements it is checked on, does not hold. *Handling* an inconsistency is done according to a chosen handling strategy, executing a series of *actions*, which can include correcting the models and their elements, changing or relaxing the consistency conditions [12] or tolerating or ignoring the inconsistency [13] (for example to postpone its resolution to a later date).

Our conceptual framework reflects this trinity and is composed of three main components: *consistency specification*, *inconsistency detection* and *inconsistency handling*, as illustrated schematically in Figure 1. A consistency specification defines the consistency conditions, expressed in some condition language(s), and the models (and their elements), expressed in some modelling language(s), that should respect these conditions. To detect inconsistencies, the consistency conditions are checked on (a subset of) the models and expressed in terms of the elements and language elements which they are composed of. *Inconsistency handling* amounts to changing the conditions and/or models. How and what actions exactly are chosen and executed, depends on the chosen inconsistency handling strategy.

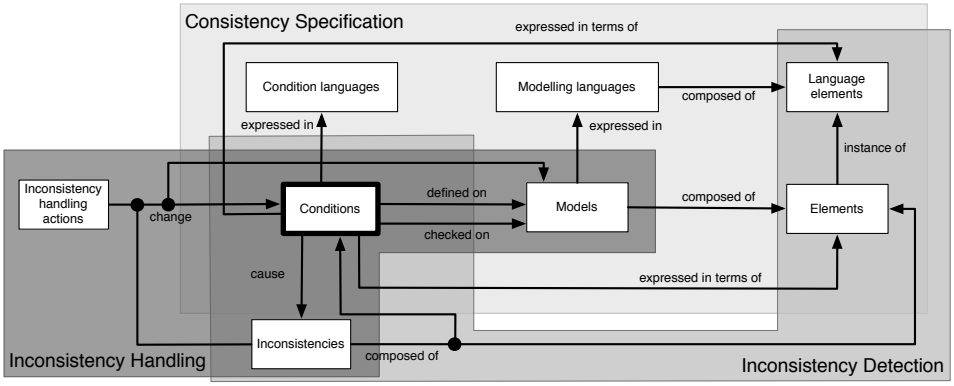
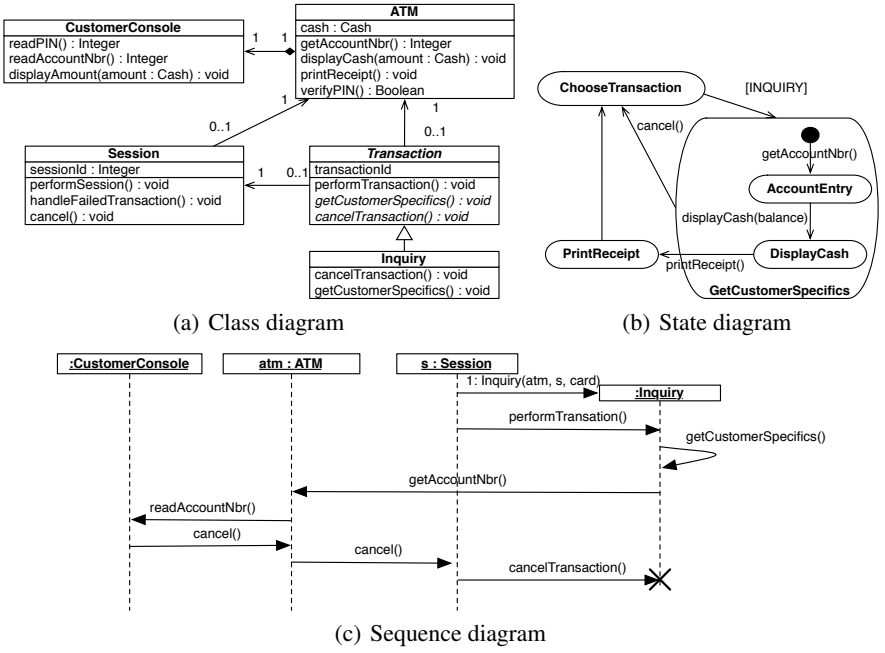


Fig. 1. Overview of the inconsistency framework.

2.2 Illustrative example

Before passing to our formalisation, we introduce a concrete instantiation on which we will illustrate the different components of the framework. Van Der Straeten *et al.* [8, 14] present an inconsistency specification and detection approach for UML models. The approach focuses on three kinds of UML diagrams: class diagrams, state machines and sequence diagrams. It supports specification and detection of a variety of inconsistencies that can be observed between those diagrams. The consistency conditions in this approach are expressed as rules in Description Logics [15]. The approach is elaborated in Section 3.2.

Figure 2 shows a class diagram, a state machine diagram and a sequence diagram that describe parts of the design of an automatic teller machine (ATM). Whereas the class diagram shows the different classes involved and how they are structurally interrelated, the state machine diagram details the specific states and transitions in case of an inquiry transaction, and the sequence diagram shows the scenario in case the inquiry transaction gets cancelled after the account number of the account to inquire is asked.



(a) Class diagram (b) State diagram

(c) Sequence diagram

Fig. 2. Examples of class, state machine and sequence diagrams of an ATM.

A possible consistency condition that we may want to express over such interrelated diagrams is the fact that every operation (like cancelling a transaction) specified on a transition in a state machine diagram or on a message in a sequence diagram, needs to be defined in the corresponding class. In [8], inconsistencies of this type are

classified as "dangling feature references" (referred to as *DFR* in the remainder of the paper, cf. Section 3.2). In the example of Figure 2, this condition is violated because the `cancel()` operation called on a transition in the state diagram and sent to an instance of the `ATM` class in the sequence diagram is not defined in the class `ATM`. As explained in [7, 8], each consistency condition can be expressed in either the Description Logic $\mathcal{SHIQ}(\mathcal{D}^-)$ or in the Description Logic query language $nRQL$ [16]. In particular, the condition *DFR* can be expressed in $nRQL$ as (taken from [7]):

```
(retrieve (?op ?c)
 (or (and (?m Message) (?m ?mend receiveEvent)
           (?mend ?ev OccurrenceSpecificationeventEvent)
           (?ev ?op ReceiveOperationEventoperationOperation) (?op Operation)
           (?mend ?l covered) (?l ?cend LifelinerepresentsConnectableElement)
           (?cend ?c type) (?c Class) (neg (?c ?op ClassownedOperationOperation)))
      (and (?stm ProtocolStateMachine) (?stm ?r region) (?r Region)
          (?r ?t  transition) (?t ProtocolTransition) (?t ?op referred)
          (?op Operation) (?c Class) (?stm ?c BehaviorcontextBehavioredClassifier)
          (neg (?c ?op ClassownedOperationOperation)))))
```

This query uses variables (symbols beginning with "?"), concepts like `Message`, `Operation`, `Region` and roles like `covered` and `receiveEvent` linking concepts together. These concepts and roles are obtained from automatically translating the UML metamodel into the DL $\mathcal{SHIQ}(\mathcal{D}^-)$.

In the next subsection, we will define the conceptual framework more formally, using this running example to explain and illustrate the different definitions introduced.

2.3 Formal definition

A formal backbone is a mandatory step in the unification of existing inconsistency management approaches. It provides a rigourous and unequivocal basis for the understanding of, comparison of and reasoning about such approaches. Formally, the inconsistency framework can be defined as follows:

Definition 1 (Inconsistency Framework $\mathcal{F}$). *Our* inconsistency framework $\mathcal{F}$ *is the set of all tuples* (s, d, h) *where:*

- $s \in \mathcal{S}$ *is a consistency specification;*
- $d \in \mathcal{D}$ *is an inconsistency detection specification;*
- $h \in \mathcal{H}$ *is an inconsistency handling specification.*

We will refer to each such tuple $f \in \mathcal{F}$ *as an* inconsistency management approach.

The definitions of $\mathcal{S}$, $\mathcal{D}$ and $\mathcal{H}$ are given in the remainder of this section.

Consistency specification The specification of consistency $s \in \mathcal{S}$ is centered around the definition of consistency *conditions* which must be respected by the different *models* in order for them to be consistent. As shown in Figure 3, every condition is expressed in terms of the *elements* of one or more models and in terms of the elements of their modelling languages. Every consistency condition is expressed in some *condition language*, which can be a logic language, a constraint language, an imperative language or

another executable language. The consistency conditions can be expressed in different languages. Every model is expressed in a *modelling language*, which can be a general-purpose modelling language, a domain-specific one or even a programming language. Like conditions, models can be expressed in different languages (e.g. UML, Java and SQL).

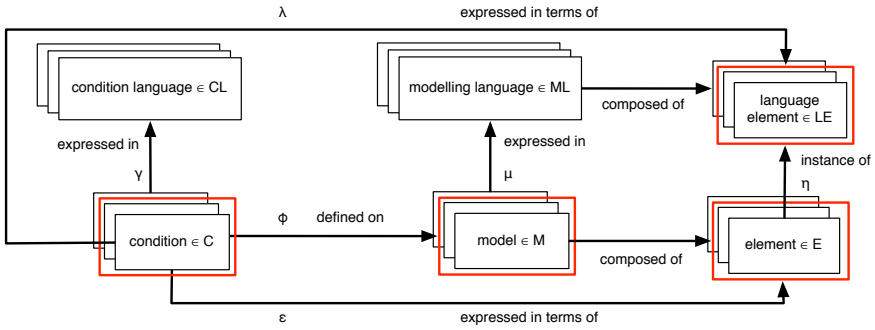


Fig. 3. Consistency specification ($\mathcal{S}$).

Example 1. The consistency specification s_{UML} of the framework instantiation $f_{UML} = (s_{UML}, d_{UML}, h_{UML})$ corresponding to the example of subsection 2.2 is defined as:

- The set M_{UML} of models considered for detecting inconsistencies. The three models presented in Figure 2 give $M_{UML} = \{m_c, m_t, m_s\}$ where m_c is the class diagram of Figure 2(a), m_t is the state machine diagram of Figure 2(b) and m_s is the sequence diagram of Figure 2(c).
- The set ML_{UML} of modelling languages in which these models are expressed. In the case of our example, this is the set $\{UML_{class}, UML_{state}, UML_{sequence}\}$ of sublanguages of UML corresponding to the different kinds of models considered. For example, UML_{class} is the sublanguage of UML describing the syntax and semantics of UML class diagrams. Alternatively, we could have opted to consider the UML as a single language which is the union of all its sublanguages corresponding to the different kinds of diagrams supported by UML.
- The set C_{UML} of consistency conditions defined over those models. An example of such a condition was given in Section 2.2. This condition as well as others described in [8] express possible domain-independent inconsistencies that can occur within and between UML sequence, UML class and UML state machine diagrams. The fact that these conditions are domain independent implies that they are expressed in terms of language elements only and do not refer to concrete model elements. For example, the $nRQL$ query shown earlier on does not *directly* refer to actual classes like `ATM` or concrete operations like `cancel`. This does not imply that model-specific conditions cannot be expressed. In that case, the condition

would also refer to concrete model elements, e.g., a condition stating that every class that specialises the `Transaction` class should also implement an operation `cancelTransaction`.
- The set CL_{UML} of languages in which those conditions are expressed is defined as $\{\mathcal{SHIQ}(\mathcal{D}^-), nRQL\}$. As explained in Section 2.2, each condition is expressed in either the description logic $\mathcal{SHIQ}(\mathcal{D}^-)$ or in the query language $nRQL$.

This brings us to our formal definition of a consistency specification.

Definition 2 (Consistency Specification $s \in \mathcal{S}$). *A consistency specification $s \in \mathcal{S}$ is a tuple $(C, CL, M, ML, \gamma, \mu, \Phi)$ where:*

- *C is a set of conditions;*
- *CL is the set of condition languages in which the conditions are expressed;*
- *M is a set of models;*
- *ML is a set of modelling languages in which the models are expressed;*
- *$\gamma : C \rightarrow CL$ is a total surjective function determining for each condition the language in which it is expressed;*
- *$\mu : M \rightarrow ML$ is a total surjective function determining for each model the language in which it is expressed;*
- *$\Phi : C \rightarrow \mathcal{P}(\mathcal{P}(M))$ is a total function defining for each condition the models on which it is defined.*

Example 2. Revisiting the consistency specification s_{UML} of example 1 we can now say that it is the tuple $(C_{UML}, CL_{UML}, M_{UML}, ML_{UML}, \gamma_{UML}, \mu_{UML}, \Phi_{UML})$ where:

- $\gamma_{UML} : C_{UML} \rightarrow CL_{UML}$ specifies for each condition whether it is expressed in either the description logic $\mathcal{SHIQ}(\mathcal{D}^-)$ or in $nRQL$.
- $\mu_{UML} : M_{UML} \rightarrow ML_{UML}$ is a function which maps the class diagram m_c to UML_{class}, the state machine m_t to UML_{state} and the sequence diagram m_s to $UML_{sequence}$.
- $\Phi_{UML} : C_{UML} \rightarrow \mathcal{P}(\mathcal{P}(M_{UML}))$. This calls for clarification. Intuitively, Φ maps a condition to the *set* of models on which it is defined, which is why we need at least a powerset. In our case, $M_{UML} = \{m_c, m_s, m_t\}$ so $\mathcal{P}(M_{UML}) = \{\{\}, \{m_c\}, ..., \{m_c, m_s\}, ..., \{m_c, m_s, m_t\}\}$. Our ***DFR*** consistency condition of Section 2.2, however, states that every operation specified on a transition in a state machine diagram or on a message in a sequence diagram, needs to be defined in the corresponding class in the class diagram. That is a condition over all possible combinations of a class diagram on the one hand and a sequence diagram or state machine diagram on the other hand. In other words, it is a constraint over the set $\{\{m_c, m_s\}, \{m_c, m_t\}\}$ which is not in $\mathcal{P}(M_{UML})$ but in $\mathcal{P}(\mathcal{P}(M_{UML}))$. More complex conditions could require an even more complex range for Φ. However, as stated before, we conceived our formalism in a bottom-up way. For each of the four different inconsistency management approaches which inspired the formalism, having a powerset of a powerset as target largely sufficed.

Definition 2 does not define the concept of *model* nor the concept of *modelling language*. Typically, a model is composed of a set of *elements*, each element being a

basic building block for model design. For instance, the UML class diagram in Figure 2(a) consists of elements like the class `ATM` and the property `cash`. In addition, the model defines links between its different elements. For instance, the nesting of `cash` inside `ATM` can be represented as a link of type `ownedAttribute` between `ATM` and `cash`. However, since the `ownedAttribute` association is an important model element in its own right, we can also see it as a model element linked to each of the elements `ATM` and `cash`. Similarly, a modelling language can be seen as a set of *language elements*, where a language element is a basic language concept. For instance, the language elements of UML class diagrams are, among others, `Class`, `Operation`, `MultiplicityElement`, `ownedAttribute`, `owneOperation` connected to each other as defined in the UML metamodel.

Seen this way, the elements and their connecting links that belong to either a modelling language or a model, define a graph which we will call an *element graph*.

Definition 3 (Element Graph G). *For a given set V of elements, and a given set A of directed links (x,y) between these elements, where $x, y \in V$, we define the* element graph *as the directed graph $G = (V, A)$. We use the notation $vertices(G) = V$ to get the vertices of the graph.*

Our sole purpose in defining models and modelling languages as graphs of elements is to provide a minimalistic formal representation of the elementary building blocks of which such models or modelling languages consist. Having such an abstract representation is essential to formally specify inconsistency detection and handling, in particular to reason at the proper level of detail about elements involved in inconsistencies. Note that this by no means restricts or imposes concrete instantiations of our framework to actually represent their models and modelling languages as graphs. Element graphs are used here merely to formally represent the elements being manipulated by the specifications, as opposed to prescribing the transformation of languages and models into a generic mathematical structure (like in [17]).

Based on this definition, we can now define the graph G_M of all model elements. We deliberately put all model elements in a single graph because of the possible occurrence of shared elements. For example, a model element representing a class `ATM` may occur both in a class diagram and in a sequence diagram. Such a shared element will be represented as a single element in the model element graph G_M. This implies that the set M of all models does not define a partition over G_M. However, we do require that G_M contains only model elements occurring in at least one model of M.

Definition 4 (Model Element Graph G_M). *G_M is the element graph (E, A_E) where E is the set of all model elements and A_E the set of arcs between them. Every model $m \in M$ defines a subgraph of G_M such that $\bigcup_{m \in M} vertices(m) = E$.*

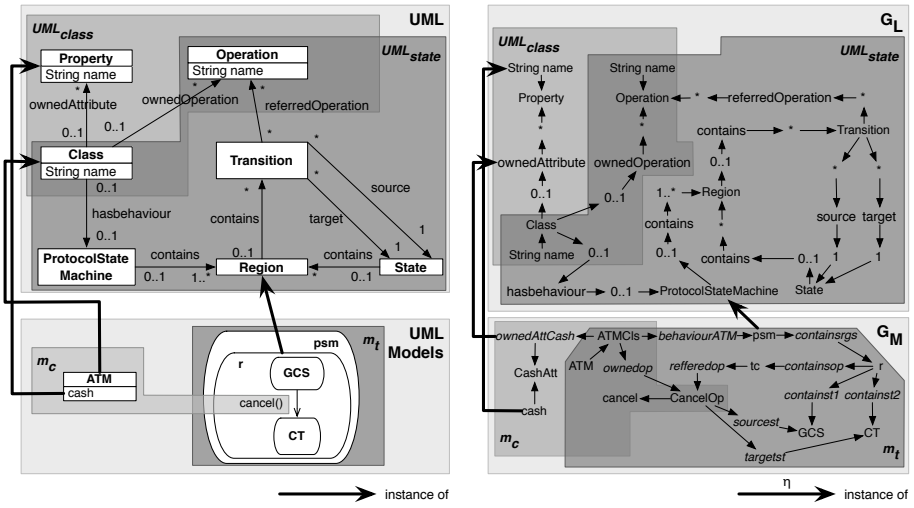
The graph G_L of all elements of modelling languages in ML is defined analogously:

Definition 5 (Language Element Graph G_L). *G_L is the element graph (LE, A_{LE}) where LE is the set of all language elements and A_{LE} the set of arcs between them. Every modelling language $l \in ML$ defines a subgraph of G_L such that $\bigcup_{l \in ML} vertices(l) = LE$.*

Evidently the model and language element graph are not unrelated. We define the relation between model elements and the language elements of which they are a linguistic instance [18] as follows:

Definition 6 (Linguistic instance of η). *$\eta : E \rightarrow LE$ is the total function determining the linguistic instance of which language element each model element is.*

Example 3. For our framework instantiation f_{UML}, the language element graph G_L corresponds to part of the UML metamodel. The model element graph G_M contains all model elements of the class, state and sequence diagrams of Figure 2. Figure 4 illustrates this on a very small subset of the class and state diagram of Figure 2, along with the respective subset of their metamodels. The model and language element graphs that correspond to, respectively, the models and metamodels of Figure 4(a), are shown in Figure 4(b). Note how, in G_L, language elements like `Class` and `Operation`, which are shared between UML_{class} and UML_{state}, are represented as single language elements in the graph. Similarly, model elements like the `cancel` operation, which are shared between different diagrams, appear as a single element in G_M. The figure also shows a few instance-of relations between model elements and their corresponding language elements. In order not to clutter the figure, only a limited set of instance-of relations is shown. Of course, in reality, η defines a mapping for every element of G_M to its corresponding element in G_L.



(a) UML language and models (b) Language and model element graphs

Fig. 4. The left part depicts a very limited subset of the UML metamodel for class diagrams and state machines and a sample of the related models from the illustrative example. The right part shows the corresponding elements as part of the language graph G_L and the model graph G_M.

Equipped with these definitions of elements and language elements, we are now set to refine the definition of a *condition*. A condition $c \in C$ can be defined on a set of model elements and/or a set of modelling language elements. Without loss of generality, a condition can involve the models only (e.g. a condition stating that a given model cannot be empty), the modelling languages only (e.g. the *nRQL* query shown before) or both (e.g. a condition on a UML class diagram representing the Factory design pattern stating that all factory classes have `public` methods of which the name starts with `make`). A condition can span several models written in different modelling languages. Evidently, the specification of a condition should also be compatible with the function Φ which specifies, for each condition, the models on which it is defined (cf. definition 2). All this can be formalised as follows:

Definition 7 (Condition Specification $c \in C$). *Every condition $c \in C$ is defined over a set of elements and/or a set of language elements such that:*

- $\epsilon : C \rightarrow \mathcal{P}(E)$ *is the total function returning, for a given condition c, the set of model elements $\epsilon(c)$ compatible with $\Phi(c)$, i.e.:*

$$\epsilon(c) \subseteq \{ \bigcup_{m \in M_{\Phi(c)}} vertices(m) \mid M_{\Phi(c)} = \bigcup_{p \in \Phi(c)} p \}$$

- $\lambda : C \rightarrow \mathcal{P}(LE)$ *is the total function returning, for a given condition c, the set of language elements $\lambda(c)$ compatible with $\Phi(c)$, i.e.:*

$$\lambda(c) \subseteq \{ \bigcup_{m \in M_{\Phi(c)}} \eta(vertices(m)) \mid M_{\Phi(c)} = \bigcup_{p \in \Phi(c)} p \}$$

Example 4. To illustrate Definition 7, we consider again the condition DFR. The λ function returns the set of involved UML language elements like `Class`, `Operation`, `Lifeline`, `Message`, `ownedOperation`, etc. ϵ returns the empty set in this case because the condition specification does not involve any model elements. Consider, however, a condition stating that the `transactionId` of each `Transaction` needs to be composed of two digits followed by an integer number. For this condition, ϵ would return at least the `Transaction` class and the `transactionId` property.

Inconsistency detection As illustrated in Figure 5 the inconsistency detection activity of the framework returns a set of *inconsistencies* raised by the evaluation of the conditions on a particular set of models. These inconsistencies can involve the *model* elements and links between those elements. The inconsistency detection does not go any step further than (1) gathering the set of inconsistencies and (2) providing accessors to the involved conditions and elements allowing to understand and reason about these inconsistencies. Observe that no link is kept to the *language* elements involved in the violation of the condition. This is because we assume the language definition to be stable and not the source of the inconsistency. Also, one can always gain access to the involved language elements indirectly, either via the involved model elements (with the η-relation), or via the conditions (with the λ-relation).

Before elaborating further on inconsistency detection, we need to define what an *inconsistency* is.

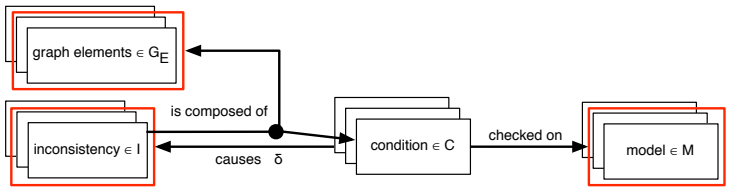


Fig. 5. Inconsistency detection ($\mathcal{D}$).

Definition 8 (Inconsistency $i \in I$). *An inconsistency $i \in I$ is a tuple (c, IE, IA) where:*

- *$c \in C$ is the violated condition;*
- *$IE \subseteq \mathcal{P}(E)$ is the set of model elements involved in the inconsistency, where $IE \subseteq \epsilon(c) \cup \{e | \eta(e) \in \lambda(c)\}$.*
- *$IA \subseteq \mathcal{P}(A_E)$ is the set of links between model elements of which at least one is involved in the inconsistency, i.e., $\forall (x, y) \in IA \bullet x \in IE \vee y \in IE$.*

Intuitively, the constraint on IE says that the only elements involved in an inconsistency can either be those model elements that are explicitly referred to in the condition, or instances of language elements that are used in the condition. The one on IA says that the arcs involved in an inconsistency must be related to an inconsistent element.

In the definition, the set of model elements IE and the set of links IA are treated separately. An alternative would have been to return a powerset of subgraphs of G_M. The disadvantage of that approach is that it would not discern whether the elements linked together or the links are involved in the inconsistency. Based on this definition of inconsistency, the detection can defined as follows.

Definition 9 (Inconsistency Detection Specification $d \in \mathcal{D}$). *An inconsistency detection specification $d \in \mathcal{D}$ is a function $\delta : C \times \mathcal{P}(M) \rightarrow \mathcal{P}(I)$. This partial function*[5] *returns the set of inconsistencies resulting from the evaluation of $c \in C$ on a set of models M_d with the additional constraint that $M_d \in \Phi(c)$. For Definition 8 to be correct, we also have to enforce that the elements in IE belong to M_d. However, we don't formalise it here for reason of succintness.*

Example 5. The function δ is implemented in f_{UML} by using RacerPro, a DL reasoning engine. Evaluation of the $nRQL$ query corresponding to DFR shown in Section 2.2 contains the tuple $((?op\ cancel)(?c\ ATM))$. This tuple returns the set of model elements IE involved in the particular inconsistency. In this case the set IE is limited to the operation `cancel` and the class `ATM`. Remark that this set can be extended by adding the variables returned by the query (for example, also the message or transition involved could be returned by the query).

Definition 9 is purposefully not specific about *how* the detection is actually performed since this may vary significantly from one inconsistency management approach

[5] δ is partial to account for cases like run-time errors or infinite loops in the conditions.

to another and from one condition language to another. Note that $\delta(c, M_d)$ takes as second parameter an element of $\mathcal{P}(M)$. This means that δ has to be called as many times as there are model tuples to check. The reason is that the detection process should be as flexible as possible and neither compel a specific checking order nor require all models to be evaluated. For instance, in the case of our illustrative example, only a subset of the models M_{UML} may need to be checked because the modeler only finished modeling that subset. For a given instantiation, the order in which models are checked might need to be prescribed because of some constraints or simply to respect some heuristic, e.g., checking first models with higher chances to fail.

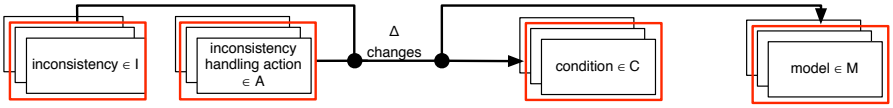


Fig. 6. Inconsistency handling specification ($\mathcal{H}$).

Inconsistency handling In inconsistency management approaches, detected inconsistencies are handled according to *actions*. Actions define the changes to be performed on models or conditions to resolve the inconsistency. Which models and conditions are selected to be handled and how they are changed is determined by *handling strategies*. A handling strategy is a combination of different actions applied to a set of selected models and conditions. A strategy is more subtle than the mere application of a series of actions leading to the resolution of inconsistencies. Possible strategies include but are not limited to resolving, deferring, ignoring and circumventing inconsistency occurrences [13]. Other examples of handling strategies are: strategies that do not introduce new (temporary) inconsistencies or strategies that do not delete model elements.

Our framework specifies the handling in terms of its impact on models and conditions based on a set of predefined actions, as depicted in Figure 6 and defined below.

Definition 10 (Inconsistency Handling Specification $h \in \mathcal{H}$). *An inconsistency handling specification $h \in \mathcal{H}$ is defined as a tuple (A, Δ) where:*

- *A is the set of possible inconsistency handling actions;*
- *$\Delta : \mathcal{P}(I) \times \mathcal{P}(A) \rightarrow \mathcal{P}(M) \times \mathcal{P}(C) \times \tau$ is an* inconsistency handling function *that derives from the detected inconsistencies and the available handling actions an* handling strategy *defining which and how models and conditions are altered.*

The inconsistency handling function Δ calls for more explanation. First, the available handling actions determine how the models and the conditions have to be changed in order to resolve the inconsistency. By using the $\mathcal{P}(A)$ we are able to propose several handling actions to resolve the inconsistencies. The $\mathcal{P}(I)$ is used here to allow the handling of several inconsistencies. For instance, one can want to resolve all inconsistencies with the same resolution pattern or related to the same model. Secondly,

the definition of Δ typically accounts for priority criteria, criticality of the models, or inter-dependencies between handling actions. Based on these parameters, the handling strategy (1) determines the models and conditions to be altered and (2) specifies a transformation function τ establishing how, based on the selected actions, they are actually altered. These new model and condition sets replace their original versions and define a revised version of f. The formal specification of τ and how the framework f is actually updated go beyond the scope of this paper.

3 Instantiations of the framework

In this section we provide several instantiations of the inconsistency framework presented above. These instantiations show how putting an existing approach in the framework can be helpful to better understand the strengths and limitations of a given approach.

3.1 Template

Below we give a brief description of the template we will use for the different instantiations of the framework.

- **Introduction:** provides a general introduction to the instantiation;
- **Illustration:** graphically illustrates the inconsistency management problem;
- **Framework instantiation:** instantiates the formal framework defined in this paper;
- **Consistency conditions:**describes the way consistency conditions are expressed;
- **Inconsistency detection:** elaborates on the inconsistency detection process;
- **Inconsistency resolution:** addresses the inconsistency resolution phase.

3.2 Instantiation 1 : Inconsistencies in Model-driven Engineering

Introduction Inconsistency management plays an important role in the context of MDE due to the following reasons.

- Models are assets in MDE. Different views of the software system are covered by different models. Because of the wide variety of models and the many relationships that can exist between them, managing these models is a very complex task and inconsistencies can arise easily.
- A model is described in a certain modelling language, e.g., the UML. The UML contains several diagram types, each described in a certain language. Each model must be legitimate with respect to the languages in which it is expressed.
- Because transformation of models is another important part of MDE, consistency between, e.g., refined models or between different evolved versions of a model is also an important issue.
- For some companies inconsistencies are more than the specification of general coherence rules between or within models. Models are regarded as inconsistent if they do not comply with specific software engineering practices or standards followed by the company.

The *Unified Modeling Language* (UML) [19] is currently the standard modelling language for object-oriented software development and well on its way to become a standard in MDE. The visual representation of UML consists of a set of different diagram types. Each diagram type is described in a certain language. Examples of such languages are class diagrams, sequence diagrams, communication diagrams and state machine diagrams. The different diagram types describe different *aspects* of a software system under study. A class diagram renders the static structure of the system. Sequence diagrams focus on the interaction of different instances of classes, i.e., objects, in a certain context. Communication diagrams describe how different objects are related to each other. Finally, state machines define how the state of a certain object changes over time. A model consists of different such diagrams. We deliberately confine ourselves to three kinds of UML diagrams: class diagrams, sequence diagrams and state machine diagrams.

Inconsistencies We base ourselves on a classification of inconsistencies that can be observed between (evolving) UML class, sequence and state diagrams (presented in [14, 8, 7]). The classification is based on two dimensions. The first dimension indicates whether structural or behavioural aspects of the models are affected. We will confine ourselves to structural inconsistencies in this paper. The second dimension concerns the level of the affected model. We differentiate between two levels, the *Specification* level and the *Instance* level. The specification level contains model elements that represent specifications for instances, such as classes, associations and messages. Model elements specifying instances, such as objects, links are at the instance level. In terms of UML diagrams, this would naturally imply that structure diagrams, such as class diagrams belong to the specification level and behaviour diagrams, such as sequence and state machine diagrams belong to the instance level. However, sequence diagrams can also belong to the specification level representing role interactions.

	Behavioural	**Structural**
Specification	invocation interaction inconsistency observation interaction inconsistency	dangling type reference inherited cyclic composition connector specification missing
Specification-Instance	specification incompatibility specification behaviour incompatibility invocation behaviour inconsistency observation behaviour inconsistency	**instance specification missing**
Instance	invocation inheritance inconsistency observation inheritance inconsistency instance behaviour incompatibility	disconnected model

Table 1. Two-dimensional inconsistency table.

Inconsistencies can occur at the **Specification** level, between the **Specification** and **Instance** level, or at the **Instance** level. The classes of observed inconsistencies are

listed in table 1. We confine ourselves to structural inconsistencies in this report because those inconsistencies can be detected and handled by Description Logic queries (cf. Chapter 6 in [7]). Consider as an example the *instance specification missing* inconsistency.

Instance specification missing occurs when an element definition does not exist in the corresponding class diagram(s). This class of inconsistencies represents among others the **dangling feature reference**, referred to as DFR in Section 2.2 and the **dangling association reference**, referred to as DAR in the remainder of the report. *Dangling (inherited) association reference* arises when a certain link (to which a stimulus (or stimuli) is related) in a sequence diagram is an instance of an association that does not exist between the classes of the linked objects (or between the ancestors of these classes).

Description Logics (DLs) [15] that are a fragment of first-order logic, are investigated as a formalism for the definition, detection and handling of inconsistencies. DLs are a family of logic languages that were primarily used for modelling database conceptual schemata. Nowadays they are often used as a foundation for ontology languages (e.g., OWL [20]). One of the most expressive DLs, i.e., $\mathcal{SHIQ}(\mathcal{D}^-)$ that is supported by the DL reasoning engine RacerPro is used to encode the UML metamodel and the UML models. For detecting and handling the structural inconsistencies of our approach, the Description Logic query language $nRQL$ [16] is used.

In our approach an inconsistency can be resolved by an inconsistency resolution rule. A generic inconsistency resolution rule has the form: `IF inconsistency X occurs in model M THEN change model M so that X is resolved`. There are typically multiple resolutions for a particular inconsistency and each one is represented by one rule. Hence, all rules pertaining to a certain inconsistency `X` have the same expression `inconsistency X occurs in model M` in their conditions. The occurence of an inconsistency in a model is detected by querying the data representing the model, i.e., the model elements. A certain state of the model attests to the presence of a particular inconsistency.

A rule's conclusion states how to resolve the detected inconsistency. It consists of a sequence of statements, where each statement is responsible for either adding data to the model or removing data from the model. As such, the model elements are rearranged so that the inconsistency is resolved. However, in order for a certain inconsistency resolution to be applicable, some model elements typically need to be present or in a particular configuration. Therefore, this is also checked in the condition of the rule, after checking the occurrence of the inconsistency.

Illustration Figure 7 represents how the approach is fitted into the proposed framework. The UML metamodel is presented by $\mathcal{SHIQ}(\mathcal{D}^-)$ terminological expressions. UML models are represented as assertional statements, i.e., logical constants and how these constants are related through relations defined in the terminological part. Structural inconsistencies are expressed as logical queries. The queries are written in $nRQL$. Queries representing the inconsistencies use the concepts defined in the DL Tbox representing the UML metamodel but are executed by the RacerPro reasoning engine on the Aboxes representing the UML models.

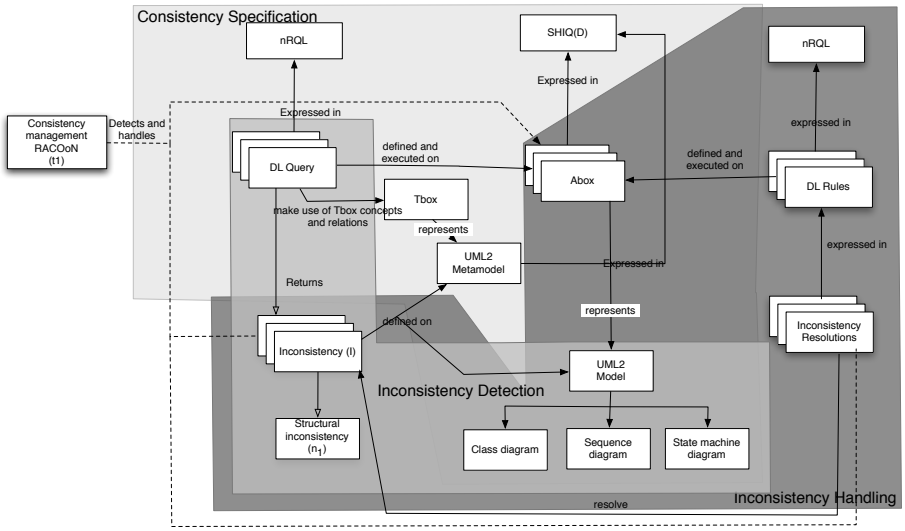


Fig. 7. Framework instantiation for UML inconsistency management using DLs

Regarding inconsistency handling, we focus on *resolution actions* that modify models in order to resolve inconsistencies. We use the term *inconsistency resolution* to indicate a set of resolution actions that resolve a certain inconsistency.

Framework instantiation f_{UML} refers to the framework instantiation of the approach presented in this section. The consistency specification s_{UML} of the framework instantiation $f_{UML} = (s_{UML}, d_{UML}, h_{UML})$ is defined as:

- The set M_{UML} of models considered for detecting inconsistencies.
- The set ML_{UML} of modelling languages in which these models are expressed. In the case of our example, this is the set $\{UML_{class}, UML_{state}, UML_{sequence}\}$ of sublanguages of the UML corresponding to the different kinds of models considered. For example, UML_{class} is the sublanguage of the UML describing the syntax and semantics of UML class diagrams. Remark that there is no unique way to create a framework instantiation for a given inconsistency management approach. For example, we could have regarded UML as a single language with different types of models. That choice, however, has an impact when comparing approaches. An interesting point of comparison between inconsistency management approaches for UML is the different types of UML models they can deal with. However, when regarding UML as a single modelling language that encompasses all models, this difference becomes less apparent.
- The set C_{UML} of consistency conditions defined over those models. An example of such a condition was given above. This condition as well as others described in Table 1 express possible domain-independent inconsistencies that can occur within and between UML sequence, UML class and UML state machine diagrams. The

fact that these conditions are domain independent implies that they are expressed in terms of language elements only and do not refer to concrete model elements. For example, the $nRQL$ query shown earlier on does not *directly* refer to actual classes like `ATM` or concrete operations like `cancel`.
- The set $CL_{UML} = \{nRQL\}$ of languages in which those conditions are expressed. As explained, each structural consistency condition of Table 1 is expressed in the query language $nRQL$.
- $\gamma_{UML} : C_{UML} \rightarrow CL_{UML}$ specifies for each condition that it is expressed in $nRQL$.
- $\mu_{UML} : M_{UML} \rightarrow ML_{UML}$ is a function which maps class diagrams to UML_{class}, state machine diagrams to UML_{state} and sequence diagrams to $UML_{sequence}$.
- According to the definition, $\Phi_{UML} : C_{UML} \rightarrow \mathcal{P}(\mathcal{P}(M_{UML}))$.

Consistency Specification Example conditions belonging to C_{UML} expressing a dangling association reference are:
The association typing the connector is not an element of the model or there is no association typing the connector.

```
(retrieve(?l ?assoc ?m)  (or (and (?l instancespecification)
                         (neg (?l (has\_known\_successor classifierspec))))
                         (and (?l ?assoc classifierspec) (?assoc association)
                         (not (?assoc ?m member)))))
```

The association typing the connector does not exist between the classes of the objects connected through the connector.

```
(retrieve (?c ?assoc cl) (and (?c connector) (?c ?assoc associationtype)
                  (?c ?cl base) (?cl class)  (?end ?assoc owningassociation)
                  (not (?end ?cl definedType)) (not (?supercl ?cl general))
                  (?end ?supercl definedType)))
```

For our framework instantiation f_{UML}, the language element graph G_L corresponds to part of the UML metamodel that describes class, sequence and state diagrams. The model element graph G_M contains all model elements of the models considered (cf. Figure 4).

Every condition $c \in C_{UML}$ is defined over a set of elements and/or a set of language elements such that:

- $\epsilon_{UML} : C_{UML} \rightarrow \mathcal{P}(E_{UML})$ is the total function returning, for a given condition c, the set of model elements involved. In our approach ϵ_{UML} returns the empty set.
- $\lambda_{UML} : C_{UML} \rightarrow \mathcal{P}(LE_{UML})$ is the total function returning, for a given condition c, the set of language elements $\lambda_{UML}(c)$ compatible with $\Phi_{UML}(c)$. The λ_{UML} function for the condition DAR returns the set of involved UML language elements like `Class`, `Association`, `Connector`, `owningAssociation`, `associationType`, etc. In this approach, the UML metamodel is translated to a DL Tbox and consistency conditions are written using concepts defined in the DL Tbox representing the UML metamodel. Therefore, we refine the definition of the λ_{UML} function as follows: $\lambda_{UML} = \kappa \circ \lambda'_{UML} : C_{UML} \rightarrow \mathcal{P}(T_{DL}) \rightarrow$

$\mathcal{P}(LE_{UML})$ where T_{DL} represents the set of roles and concepts representing UML metamodel concepts. The function $\kappa : \mathcal{P}(T_{DL}) \rightarrow \mathcal{P}(LE_{UML})$ represents the mapping from these roles and concepts towards the corresponding UML metamodel concepts.

Inconsistency detection The function δ_{UML} is implemented in f_{UML} by using RacerPro, a DL reasoning engine. Evaluation of the above presented $nRQL$ queries over a set of models selected by the user can result in tuples of which each returns the set of model elements IE involved in the particular inconsistency.

Inconsistency Handling The approach focusses on inconsistency management, which means that inconsistencies are tolerated in the Aboxes and as such in the UML models. Our focus is to detect inconsistencies over the model. However detecting an inconsistency can be a false positive, i.e., the queries are manually written and can contain errors. Our approach does not take any handling actions over consistency conditions into consideration.

An inconsistency occurrence can be resolved by possibly a set of resolutions. Each resolution is expressed as a DL rule in the nRQL language. Resolutions are applied on the DL Aboxes representing the UML model. The following two rules represent possible resolutions for the DAR inconsistency. The condition of each rule includes a $check - DAR$ statement representing the query verifying the DAR condition. Other statements included in the condition of the rule check whether certain model elements necessary to resolve the inconsistency, are available in the model. The first rule below expresses the creation of a new association from the target class to the source class, whereas the second rule uses the existing association if this exists.

```
(firerule
  (and (check-DAR ?assoc ?m)
       (?m ?con connectorr)
       (?m ?mendsend sendEvent)
       (?mendsend ?lifelinesend coveredsub)
       (?lifelinesend ?connectableelsend represents)
       (?connectableelsend ?classsend base)
       (?m ?mendreceive receiveEvent)
       (?mendreceive ?lifelinereceive coveredsub)
       (?lifelinereceive ?connectableelreceive represents)
       (?connectableelreceive ?classreceive base)
       (user-option-addAssoc ?assocname))
(  (related (new-ind assoc ?assocname) ?assocname name)
  (related (new-ind assoc ?assocname)
           (new-ind end ?classsend) memberend)
  (related (new-ind assoc ?assocname)
           (new-ind end ?classreceive) memberend)
  (related ?class (new-ind end ?classsend) ownedattribute)
  (related ?class2 (new-ind end ?classreceive)
                ownedattribute)
  (related ?con (new-ind assoc ?assocname)
                associationtype)
  (forget-role-assertion ?con ?assoc associationtype))
 )

(firerule
  (and (check-DAR ?m ?assoc)
       (?m ?con connectorr)
       (user-option-useAssoc ?assocuser)
  )
```

```
  (
  (related ?m (new-ind connector ?assocuser) connectorr)
  (related (new-ind connector ?assocuser) ?assocuser associationtype)
  (forget-role-assertion ?m ?con connectorr)
  )
)
```

We denote the set of possible inconsistency resolution rules presented in our approach as A_{UML}. The inconsistency handling function $\Delta_{UML} : \mathcal{P}(I) \times \mathcal{P}(A_{UML}) \rightarrow \mathcal{P}(M_{UML}) \times \tau$ derives from the detected inconsistencies and the available handling actions a *handling strategy* defining which and how models are altered. Currently this is a manual process. However based on an analysis of the possible relationships between resolution rules [21] we are trying to automate this process.

3.3 Instantiation 2: Co-evolving Source Code and Structural Design Regularities

Introduction Keeping the source code of a program consistent with the design regularities and programming conventions that govern its design is an important issue in software development and evolution. Many tools exist for defining and checking such design regularities (e.g., commonly accepted best practice patterns, code conventions, etc.). Whereas some tools are restricted to checking a fixed set of built-in regularities, others are more generic and can allow their users to define and verify customised regularities. In particular, the technique of intensional views [22] and their associated tool suite IntensiVE [9] belong to the latter category. In this technique, regularities are defined intensionally with a declarative language (in terms of sets of software artefacts and binary structural relationships between those sets) and can be checked against the source code. Inconsistencies are detected when the source code does not respect the defined regularities. Inconsistencies need to be fixed manually, either by fixing the source code, refining the rules defining the regularities, or by tagging some of the irregularities as accepted exceptions to the rule.

Illustration An example of a well-known design regularity that could be defined and checked against the code is the *factory design pattern* [23]. This pattern is typically implemented with *factory* objects that create certain kinds of objects, called *products*, implementing a particular interface, as illustrated in Figure 8. The pattern requires that all factories belonging to a same family must be able to create the same set of products. A possible inconsistency of this requirement would occur if there exists a factory that cannot instantiate some product. An example of such an inconsistency is shown in Figure 9: the `AlternativeFactory` cannot create objects of class `Product1`.

With the intensional views approach, a simplified version of the factory design pattern implementation of Figure 8 could be defined in terms of two intensional source-code views V_{Fact} and V_{Prod} and a relation $C_{instantiates}$ over these views, as shown in figures 8 and 9. As stated above, both views and relations are defined by means of rules in a declarative language. The language is SOUL [24], a Prolog-dialect that can reason over actual source code artefacts. For example, the view V_{Fact} of all factory classes could be defined in terms of the following logic query which collects all classes in the class hierarchy with root `Factory`, but excluding that root class:

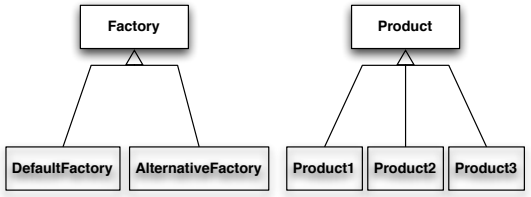


Fig. 8. An implementation of the Factory design pattern

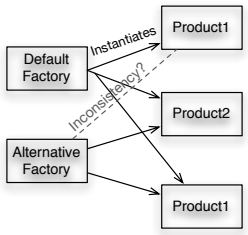


Fig. 9. An inconsistency in the implementation of the Factory design pattern

```
classBelow(?class,[Factory])
```

and the view V_{Prod} of all product classes could be defined in terms of a logic query that collects all non-abstract classes below the class `Product`:

```
classBelow(?class,[Product]),
not(abstractClass(?class))
```

The relation $C_{instantiates}$ would be a relation between V_{Fact} and V_{Prod} defined as: $\forall f \in V_{Fact} \bullet \forall p \in V_{Prod} \bullet isCreatedBy(?f, ?p)$, where $isCreatedBy$ is a logic predicate, defined as :

```
isCreatedBy(?f, ?p) if
   classHasMethod(?f, ?m),
   methodInstantiatesClass(?m, ?p)
```

In other words, this binary relation requires that for every factory class and every product class, the factory class must have a method that can create instances of the product class. Every violation of this condition is an inconsistency. For example, as illustrated in Figure 9 an inconsistency would occur if the class `AlternativeFactory` has no method that can create instances of the class `Product1`.

Framework instantiation Let $f_{IV} = (s_{IV}, d_{IV}, h_{IV})$ refer to the framework instantiation of the intensional views technique, where the consistency specification s_{IV} is defined as a tuple: $(C_{IV}, CL_{IV}, M_{IV}, ML_{IV}, \gamma_{IV}, \mu_{IV}, \Phi_{IV})$. More specifically, we can define each of the components of this consistency specification as follows:

– CL_{IV} is the condition language[6] in which we describe our structural regularities. An example of such a regularity was already presented above. In general, a structural regularity is described as a logic expression of the form

$$Q_1 x \in V_1 \bullet Q_2 y \in V_2 \bullet p(x, y)$$

[6] To be precise, it's a singleton set consisting of a single condition language.

where Q_1 and Q_2 are set quantifiers (like $\forall$, $\exists$, $\exists!$), V_1 and V_2 are intensional source-code views (declared as logic queries in the logic language SOUL) and *p* is a binary logic predicate written in the SOUL language.

- C_{IV} is a set of structural regularities we want to verify over the source code. In our example above, there was only such condition so C_{IV} would be the singleton set consisting of the sole condition $C_{instantiates}$ described above.
- The intensional views technique reasons over the source code of programs in some programming language (like Smalltalk, Java, C or Cobol). ML_{IV} thus corresponds to the programming language of the program we want to reason about.[7] However, we don't reason over the source code directly, but by using a logic library of predicates that can reason over that source code. Therefore, ML_{IV} corresponds to that library of predicates (reasoning over programs in a different programming language only requires changing the logic library).
- M_{IV} is the source code of the program we want to reason about. How that code is actually accessed is determined by the logic library defined in ML_{IV}.
- In our case, $\gamma_{IV} : C_{IV} \rightarrow CL_{IV}$ is a trivial function because we only have one condition language, so any condition is mapped to that condition language.
- $\mu_{IV} : M_{IV} \rightarrow ML_{IV}$ is trivial too, since we consider only one programming language.
- $\Phi_{IV} : C_{IV} \rightarrow \mathcal{P}(\mathcal{P}(M_{IV}))$ maps every condition to the source code artefacts it is defined over. For example $\Phi_{IV}(C_{instantiates})$ is the set consisting of the classes `DefaultFactory`, `AlternativeFactory`, `Product1`, `Product2` and `Product3` as well as the instance creation relations between these classes.

Figure 10 graphically illustrates the instantiation of the framework for the inconsistency management of design and code using IntensiVE. Consistency conditions are represented as constraints on software views. These software views are intensionally defined over source code elements such as classes, methods, etc. The language in which both views and their constraints are represented is SOUL.

Consistency conditions According to our definition of consistency specification, ϵ is a function that will return the relevant source code elements presented in a source code representation, that are intensionally defined by the software views used in a condition. In our example, the code elements are the class *Product*, the class *Factory*, and the classes below them. λ is a function returning for every source code element, the language element to which it belongs to (e.g., for the code element *Factory*, its corresponding language element is *Class*).

Inconsistency detection An inconsistency will be detected if a specified relationship does not hold for certain source code elements belonging to the Intensional Views being checked. According to our definition, the inconsistency detection specification is a function mapping a consistency definition and a set of models, to an inconsistency: $\delta : C \times \mathcal{P}(M) \rightarrow \mathcal{P}(I)$

We have to not that in our instantiation of our framework, $\mathcal{P}(M)$ will be reduced to a singleton model M, since only one model is implied. In the context of our example, the inconsistencies are found in the following way:

[7] To be precise, it's a singleton set consisting of a single language.

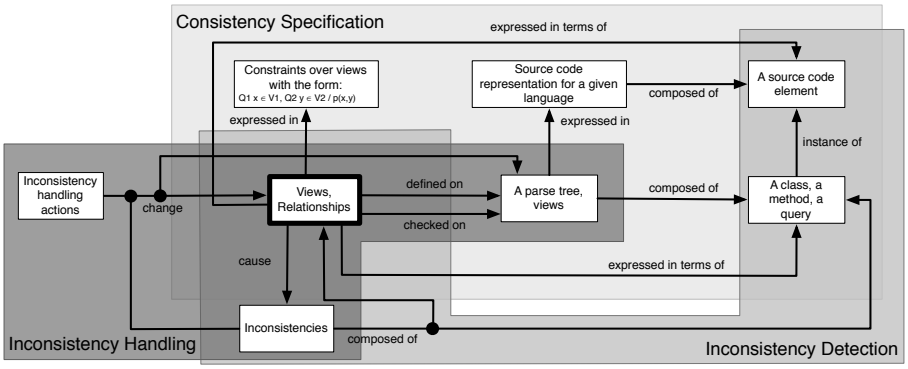


Fig. 10. Framework instantiation for code-design inconsistency management

- evaluate the extents of the views V_{Fact} and V_{Prod} referred by the predicate $isCreatedBy$;
- evaluate the predicate $isCreatedBy$ in the cross product of these two views;
- report all the the elements $f \in V_{Fact}$ and $p \in V_{Prod}$ for which the predicate $isCreatedBy$ does not hold;

Inconsistency resolution In the case that inconsistencies are detected, one of the following actions has to be taken

- modify one of the views definition;
- modify the code referred by the views;
- modify the conditions over the views (i.e., either changing the predicates or the quantifiers)

3.4 Instantiation 3: inconsistencies between data models and queries over those models (in the context of database schema evolution)

Introduction Analyzing the impact of database schema[8] evolutions on associated programs can be also considered as another example of consistency management problem. Indeed, the impact of schema transformations can be defined as the set of database queries becoming inconsistent with respect to the new schema. Recent studies show that schema evolutions may have a dramatic impact on queries, reaching up to 70% query loss per schema version [25].

Illustration Figure 11 provides an example of a SQL query becoming inconsistent due to the renaming of table `CUSTOMER` into table `CLIENT`. Figure 12 further illustrates how the general problem of checking the consistency of a query against its database schema can be considered as an instance of our inconsistency management framework. The consistency conditions reference the query language (SQL, COBOL,

[8] Here, data model and database schema are considered as synonyms

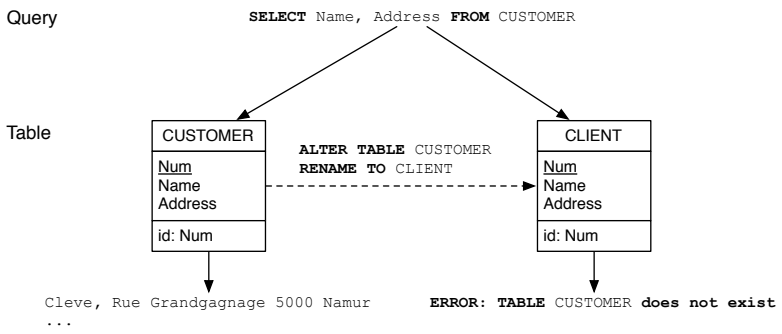


Fig. 11. A SQL query becoming inconsistent due to an evolving relational schema.

CODASYL, IMS, etc.) and the database schema metamodel. Our approach makes use of the Generic Entity-Relationship model [26] (GER) for describing database schemas. The GER model encompasses the major database paradigms (relational, network, hierarchical, ER, UML and XML models) and allows to specify database structures at different levels of abstraction (conceptual, logical and physical).

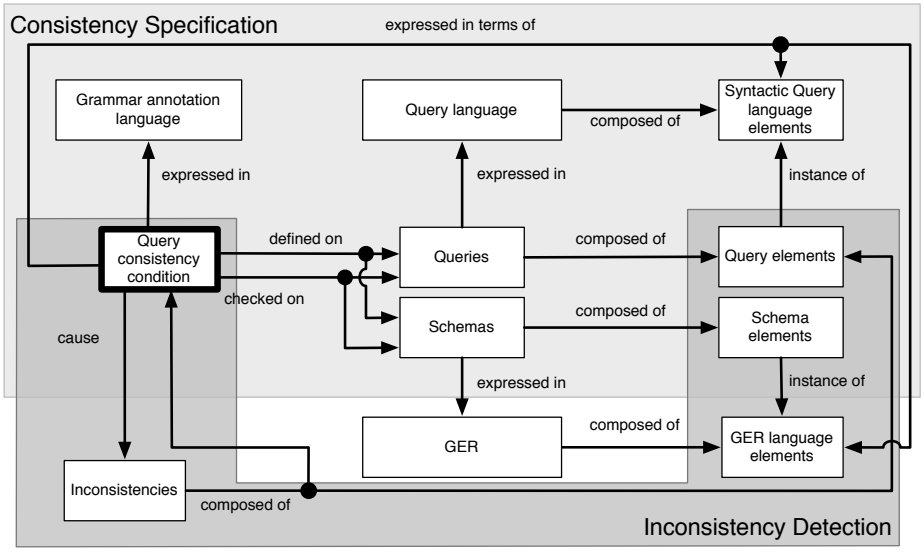


Fig. 12. Framework instantiation for query-schema inconsistency management

Framework instantiation The consistency specification s_{QS}[9] of the framework instantiation $f_{QS} = (s_{QS}, d_{QS}, h_{QS})$ is defined as:

[9] QS denotes Query-Schema

- The set M_{QS} of models (queries and schemas) considered for detecting inconsistencies.
- The set ML_{QS} of modelling languages in which these models are expressed. In this case, this is the set $\{GER, SQL, CODASYL, COBOL, \cdots\}$.
- The set C_{QS} of consistency conditions defined over those models.
- The set CL_{QS} contains a grammar annotation language, defined on top of the SDF syntax definition formalism [27].
- $\gamma_{QS} : C_{QS} \rightarrow CL_{QS}$ specifies for each condition that it is expressed in our grammar annotation language.
- $\mu_{QS} : M_{QS} \rightarrow ML_{QS}$ is a function which maps the queries to their query language and the schemas to the GER language.
- According to the definition, $\Phi_{QS} : C_{QS} \rightarrow \mathcal{P}(\mathcal{P}(M_{QS}))$. In practice, we iteratively check *a set of* queries against a *single* schema. Each checking involves a couple $(query, schema)$.

For our framework instantiation f_{QS}, the language element graph G_L contains (1) syntaxic constructs of the query grammar and (2) GER constructs (entity type, attribute, relationship types, collections, groups, etc.). The model element graph G_M contains all model elements of the models considered. For the queries, G_M is close to an abstract syntax tree. For the schemas is contains instances of the GER constructs (e.g. entity type `CUSTOMER`, attribute `NAME`, etc.).

Every condition $c \in C_{QS}$ is defined over a set of elements and/or a set of language elements such that:

- $\epsilon_{QS} : C_{QS} \rightarrow \mathcal{P}(E_{QS})$ is the total function returning, for a given condition c, the set of model elements involved. In our approach ϵ_{QS} returns the empty set.
- $\lambda_{QS} : C_{QS} \rightarrow \mathcal{P}(LE_{QS})$ is the total function returning, for a given condition c, the set of language elements $\lambda_{QS}(c)$ compatible with $\Phi_{QS}(c)$. The λ_{QS} function for the condition returns (1) the set of involved query language elements like `TableName`, `FromClause`, `ColumnName`, `GroupByClause`, etc, combined with (2) the set of GER constructs like entity type, attribute, etc.

Consistency conditions As already indicated, the consistency conditions that must hold between a query and its database schema express the relationships between the query language of interest and its underlying schema metamodel (the GER). According to our approach, the consistency conditions consists of domain-specific *grammar annotations* defined over the query language syntax. Let us consider the following simplified syntax for SQL queries:

```
SelectClause FromClause -> Query
"SELECT" Column-list -> SelectClause
"FROM" Table-list -> FromClause
{Column ','}+ -> Column-list
{Table ','}+ -> Table-list
```

We can impose several consistency conditions on the instances of syntax production `Query`. First, we need to make sure that each table name occuring in the from clause

of the query corresponds to a declared table in the database schema. Since in the GER model, a SQL table is represented as an *entity type*, this condition can be expressed as follows:

```
for each t:Table in FromClause : isAnEntityType(t)
```

This condition is not sufficient for ensuring the consistency of the query. In addition, it is required that each column name of the select clause corresponds to a column of at least one table of the from clause. In the GER model, a column is represented as an *attribute*. Thus, this second consistency rule can specified through the following second condition:

```
for each c:Column in SelectClause :
    exists t:Table in FromClause : isAnAttributeOf(c,t)
```

Inconsistency detection Based on the annotated grammar of the data manipulation language, a inconsistency detection tool is automatically derived. This tool is based on the ASF+SDF technology [28]. It takes as inputs a set of queries (i.e., instances of the query language grammar) together with the underlying schema description, and returns the set of detected inconsistencies wrt the specified consistency conditions.

The generated ASF+SDF consistency checker actually implements function δ_{QS} of our framework instantiation. It returns a set of inconsistencies, each of which is linked (1) to the violated condition and (2) the source code location of the inconsistent query.

Inconsistency handling Our QS instantiation mainly focusses on the inconsistency detection activity. As we are checking consistency in the context of impact analysis, the detected inconsistencies are to be considered as *potential*. Depending on the impact, the database manager may decide to cancel a desired schema evolution. In this case, no inconsistency handling actions must be undertaken. But in case the schema evolution is performed, the inconsistent queries have to be reexpressed against the new schema. We have proposed in [10] a co-transformational approach to schema-query co-evolution, according to which query transformations are associated to schema transformations. In this context, fully automated inconsistency handling is possible in the presence of *semantics-preserving* schema transformations.

3.5 Instantiation 4: Inconsistencies in model transformation specifications

Introduction Specifying model transformations can be done in a visual manner. The tool, MoTMoT, developed at the University of Antwerp employs UML diagrams to represent model transformations. Class diagrams and activity diagrams are used and checked if they correspond to the syntax of the UML profile for Story Driven Modeling.

Story Driven Modeling (SDM) is a model transformation language supporting an imperative control flow for graph transformations. Evaluating consistencies can be done on two levels: on the one hand, the specification level where the transformation is modeled using a mix of UML class and activity diagrams. On the other hand we have the execution level where an input model is processed and transformed resulting in an output

model. At the specification level preconditions are verified to see if the transformation is modeled according to the SDM language, at the execution level the transformation is executed and sample output models can be checked on their equivalence. For the remainder of this subsection we focus only on the specification level.

Illustration The tool MoTMoT can only parse correct SDM diagrams. For this reason we use OCL constraints to maintain a strict consistency. In MoTMoT we cannot tolerate any inconsistency and consistency resolution is then focused on fixing the elements involved in the constraint to achieve a consistency.

In this illustration we focus on just a select number of constraints, however all constraints in MoTMoT are set up in a similar manner. OCL constraints are defined on UML class diagrams and activity diagrams, so the following constraints will only involve those language elements.

- Each state in the UML activity diagram should be linked to a UML class diagram. The class diagram contains the description of the graph transformation. This is valid for all states, except when the state is a code state (containing Java code instead) or a link state (referring to another control flow):

  ```
  context MotMotActionStateFacade
  inv: isTransPrimitiveState() implies
  hasTransPrimitivePackage() = true
  ```

  So for each action state in the activity diagram and if that state is supposed to be contain a transformation diagram, we check if that state indeed has a link to a class diagram.
- All classes within a class diagram, which are part of a graph transformation, should have a type. The classes represent typed nodes, the existence of untyped nodes is prohibited. This is done with the constraint:

  ```
  context MotMotClassFacade
  inv: isPartOfTransformation() implies
  getTypeName() <> ''
  ```

  So, for each class which is part of a transformation pattern we look that they have a type attached to it.

Observe figure 13 where we show an incorrect UML activity diagram. The state named `Create List Impl` is missing a tagged value to denote it is connected to a class diagram. All other states in the diagram do have a connection with a tagged value `motmot.transprimitivepackage`, a link to the UML package containing the matching class diagram.

Framework instantiation The instantiation of the framework can be seen on figure 14. The consistency specification s_{TM}[10] of the framework instantiation $f_{TM} = (s_{TM}, d_{TM}, h_{TM})$ is defined as:

The consistency specification is defined as:

[10] TM denotes Transformation Model

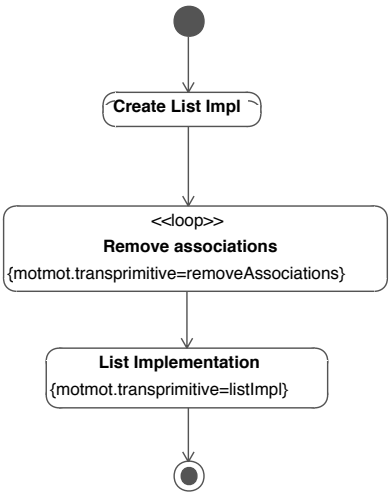


Fig. 13. Illustration of an incorrect UML Activity Diagram for MoTMoT

- The set M_{TM} of models (UML class and activity diagrams) considered for detecting inconsistencies.
- The set ML_{TM} of modelling languages in which these models are expressed. In this case, this is the set $\{UML, Java, \cdots\}$.
- The set C_{TM} of consistency conditions defined over those models.
- The set CL_{TM} is set in the OCL.
- $\gamma_{TM} : C_{TM} \rightarrow CL_{TM}$ specifies for each condition that it is expressed in OCL and $Java$.
- $\mu_{TM} : M_{TM} \rightarrow ML_{TM}$ is a function which maps class and activity diagrams to UML and $Java$ (the diagrams could contain Java code).
- According to the definition, $\Phi_{TM} : C_{TM} \rightarrow \mathcal{P}(\mathcal{P}(M_{TM}))$.

For our framework instantiation f_{TM}, the language element graph G_L contains the UML language elements for class and activity diagrams. The model element graph G_M contains all model elements of the models considered.

Every condition $c \in C_{TM}$ is defined over a set of elements and/or a set of language elements such that:

- $\epsilon_{TM} : C_{TM} \rightarrow \mathcal{P}(E_{TM})$ is the total function returning, for a given condition c, the set of model elements involved. In our approach ϵ_{TM} returns the empty set.
- $\lambda_{TM} : C_{TM} \rightarrow \mathcal{P}(LE_{TM})$ is the total function returning, for a given condition c, the set of language elements $\lambda_{TM}(c)$ compatible with $\Phi_{TM}(c)$. The λ_{TM} function for the condition returns the set of involved UML language elements like `Class`, `Association`, `ActionState`, `TaggedValue`, etc.

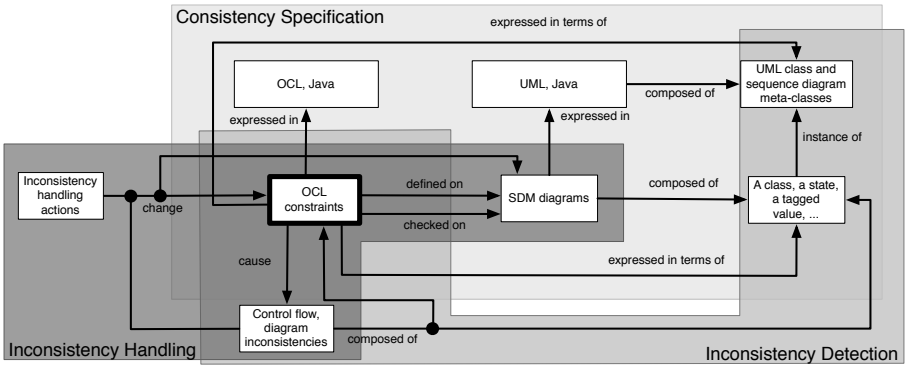


Fig. 14. Framework instantiation for Model Transformation specifications

Consistency conditions The first set of inconsistencies we want to avoid are violations of the SDM syntax. For example, a class diagram represents a graph pattern for MoTMoT. Each class represents a node in the graph pattern. Each class requires a model element type tag to know which model element is represented. Conditions are specified in the Object Constraint Language (OCL) and deal specifically with the layout of model transformation specifications. The conditions are checked on UML class diagrams. UML class diagrams have to conform to the UML language (this is usually enforced through the UML diagram tool) and the OCL conditions for MoTMoT transformation models. Structural inconsistencies are flagged and shown to the user. Due to the nature of the transformation tool, user input is required to fix inconsistencies and automatic consistency resolution is not possible.

The second set of inconsistencies involve transformation specifications where the validation process is successful, though the execution of the specification does not yield the expected output. The SDM language is not sufficiently expressive or the model transformation or the transformation has a correct syntax though the semantics do not hold any ground.

Inconsistency detection There are basically two manners to enforce consistent models representing model transformations. The UML profile for SDM restricts users in their use of UML model elements: only a subset of model elements can be used, therefore limiting the language. OCL constraints impose more syntactic conditions on the models. For example, every state in the control flow of the transformation should contain a graph rewriting. An OCL constraint could impose that states include graph rewriting diagrams.

AndroMDA is a tool which translates SDM diagrams into Java code and has an OCL validation tool built in. Before translation, a validation phase is run and if the model cannot be validated a set of violated constraints is shown, along with a comment explaining the error and possibly offering a suggestion to fix the violated constraint.

Inconsistency resolution Consistency resolution involves three stages:

- Sample models are used to verify conditions. There are correct models and models which violate one or more conditions. The test coverage should cover all present conditions, however it is impossible to verify the absence of errors. This is a caveat we have to take into account. While models could appear as consistent models, they are not valid models for the transformation engine. In this case we move onto the following stage.
- After identifying and fixing conditions, the at-this-time correct models are put through the transformation engine. If the transformation fails, this is a signal that despite the conditions, these were not sufficient to prevent a failed transformation. The solution is two-fold: adjust the current conditions to be more strict and add more conditions for a more complete validation of the models.
- After these two stages the next step is evaluation of the result the transformation engine puts out. Even if all conditions are satisfied and the model can be transformed, the result can be unexpected. In this case several resolution options are available.
  - The transformation engine could be the cause, it produces the wrong results. The transformation engine should be fixed. For this purpose additional test data is necessary for evaluation.
  - The language could lack expressiveness to capture the intended transformation. Language extensions are necessary.

4 Related Work

The different concepts and definitions constituting the proposed framework are inspired by the research background of the authors [8–11] and by several definitions and classifications of inconsistencies that can be found in the literature [2–7, 29–31]. These existing inconsistency classifications focus on the definition and classification of different inconsistency types but do not achieve a common frame of reference for inconsistency management approaches.

The *viewpoints framework* has been developed by Finkelstein *et al.* [32] and extended by Nentwich *et al.* [33] to handle inconsistencies in software development using different viewpoints. The framework focuses on the definition and usage of viewpoints and uses particular technology to detect and handle inconsistencies. In Küster [34] a general methodology for *consistency management of object-oriented behavioral models* is introduced. The approach contains generic definitions of consistency conditions, consistency concepts (a way to group consistency conditions) and consistency. However, the focus is on the methodology comprising activities and specific techniques for, e.g., automatically translating models into a semantic domain using a rule-based approach relying on graph transformation. Nuseibeh *et al.* [35, 13] introduce a *framework for managing inconsistencies*. The framework itself consists of a repository of pre-defined consistency rules and of some components, such as a component detecting the inconsistencies using the consistency checking rules, a diagnosing component, a handling component. Spanoudakis et al. [1] present a survey of inconsistency management techniques. The survey is organized along a conceptual framework that views inconsistency

management as a process composed of different activities. While the framework includes definitions of overlap relations between model elements and inconsistency in terms of these overlaps, its main focus is on the process of consistency management.

These different frameworks guided the elicitation of concepts and definitions of our framework. Yet, they all have a certain focus and use specific techniques for detection or handling of inconsistencies or define a set of concrete inconsistencies. Instead, we target a unifying framework suitable for instantiation by different communities, hence the need for the formal specification of a shared vocabulary. Our work provides a common ground to assess the complementarity of the above described approaches.

5 Conclusion and future work

In this paper, we proposed a novel conceptual framework paving the way toward the unification of existing inconsistency management approaches. The aim of our formalisation is to concisely and unambiguously define a frame of reference that can be used unequivocally by framework customizers and users to understand and compare different approaches. The benefits for implementers using the framework by making instantiations of it, are that (1) they have a common vocabulary, (2) the framework gives them a reference framework such that they do not have to start from scratch, and (3) the framework allows them to reflect on their own approach and discover their shortcomings, advantages or disadvantages.

We instantiated the framework on four different inconsistency management approaches from different domains operating at different phases of the software life-cycle. We are working on a further refinement of the formalism and an assessment of its generality by validating it on existing inconsistency management approaches found in literature.

Our aim in the near future is to use the framework to compare the instantiations and to try to come up with a taxonomy or classification of the different inconsistency management approaches based upon the observed differences and similarities.

The definition of inconsistency handling touched upon the need to represent revised versions of the model and condition sets. The necessity to deal with their evolution, and incidentally the evolution of the framework, marks the limit of the current formalisation. Neither the framework's evolution nor the reasoning about traces of changes are currently supported. Further, we assumed the modeling languages described in the framework to be stable and not a possible source of inconsistencies. This assumption needs to be dropped in case we want to incorporate language evolution in the framework. In the future, we envisage the extension of our framework in the context of evolution by adding activities like impact analysis, model co-evolution and language evolution.

Theoretical extensions of the framework will concentrate on the fosterage of automated analyses and reasonings and on the study of including evolution issues in the framework.

Acknowledgements

This research is funded by the Interuniversity Attraction Poles Programme of the Belgian State, Belgian Science Policy.

References

1. Spanoudakis, G., Zisman, A.: Inconsistency management in software engineering: Survey and open research issues. In K., C.S., ed.: Handbook of Software Engineering and Knowledge Engineering. Volume 1. World Scientific Publishing Co. (2001) 329–380
2. S.Easterbrook, A. Finkelstein, J.K., Nuseibeh, B.: Coordinating distributed viewpoints: the anatomy of a consistency check. Concurrent Engineering **2**(3) (1994) 209–222
3. Engels, G., Küster, J.M., Heckel, R., Groenewegen, L.: Towards consistency-preserving model evolution. In: IWPSE '02: Proceedings of the International Workshop on Principles of Software Evolution, New York, NY, USA, ACM Press (2002) 129–132
4. Liu, W., Easterbrook, S., Mylopoulos, J.: Rule based detection of inconsistency in UML models. In Kuzniarz, L., Reggio, G., Sourrouille, J.L., Huzar, Z., eds.: Blekinge Institute of Technology, Research Report 2002:06. UML 2002, Model Engineering, Concepts and Tools. Workshop on Consistency Problems in UML-based Software Development. Workshop Materials, Department of Software Engineering and Computer Science, Blekinge Institute of Technology (2002) 106–123
5. Sourrouille, J.L., Caplat, G.: Constraint checking in uml modeling. In: SEKE '02: Proceedings of the 14th international conference on Software engineering and knowledge engineering, New York, NY, USA, ACM Press (2002) 217–224
6. Elaasar, M., Briand, L.: An overview of uml consistency management. Technical Report Technical Report SCE-04-18, Carleton University, Ottawa, Canada (2004)
7. Van Der Straeten, R.: Inconsistency Management in Model-driven Engineering. An Approach using Description Logics. PhD thesis, Department of Computer Science, Vrije Universiteit Brussel, Belgium (September 2005)
8. Van Der Straeten, R., Mens, T., Simmonds, J., Jonckers, V.: Using Description Logic to maintain consistency between UML models. In: Proc. of the 6th Int. UML Conf. (UML'03). Volume 2863 of LNCS., Springer (2003) 326–340
9. Mens, K., Kellens, A.: IntensiVE, a toolsuite for documenting and testing structural source-code regularities. 10th Conference on Software Maintenance and Re-engineering (CSMR) (2006) 239–248
10. Cleve, A., Hainaut, J.L.: Co-transformations in database applications evolution. In Lämmel, R., Saraiva, J., Visser, J., eds.: Generative and Transformational Techniques in Software Engineering. Volume 4143 of Lecture Notes in Computer Science., Springer (2006) 409–421
11. Schippers, H., Gorp, P.V., Janssens, D.: Leveraging uml profiles to generate plugins from visual model transformations. Electr. Notes Theor. Comput. Sci. **127**(3) (2005) 5–16
12. Gorp, P.V., Altheide, F., Janssens, D.: Towards 2d traceability in a platform for contract aware visual transformations with tolerated inconsistencies. Enterprise Distributed Object Computing Conference, IEEE International **0** (2006) 185–198
13. Nuseibeh, B., Easterbrook, S.M., Russo, A.: Making inconsistency respectable in software development. Journal of Systems and Software **58**(2) (2001) 171–180
14. Simmonds, J., Van Der Straeten, R., Jonckers, V., Mens, T.: Maintaining consistency between UML models using Description Logic. L'OBJET **10**(2-3) (2004) 231–244
15. Baader, F., Calvanese, D., McGuinness, D., Nardi, D., Patel-Schneider, P., eds.: The Description Logic Handbook: Theory, Implementation and Applications, 2nd Edition. Cambridge University Press (2007)

16. Wessel, M., Möller, R.: A high performance semantic web query answering engine. In: Proc. of the Int. Workshop on Description Logics (DL'05). Volume 147 of CEUR Workshop Proceedings. (2005)
17. Boyd, M., McBrien, P.: Comparing and transforming between data models via an intermediate hypergraph data model. Journal on Data Semantics IV **3730** (2005) 69–109
18. Kühne, T.: Matters of (meta-)modeling. Soft. and Syst. Modeling **5**(4) (2006) 369–385
19. Object Management Group: Unified Modeling Language specification version 2.2. formal/2009-02-02 (March 2009)
20. W3C Recommendation: OWL Web Ontology Language Semantics and Abstract Syntax. TR/owl-semantics/ (February 2004)
21. Mens, T., Van Der Straeten, R., D'Hondt, M.: Detecting and resolving model inconsistencies using transformation dependency analysis. In Nierstrasz, O., Whittle, J., Harel, D., Reggio, G., eds.: Model Driven Engineering Languages and Systems, 9th International Conference, MoDELS 2006, Genova, Italy, October 1-6, 2006, Proceedings. Volume 4199 of Lecture Notes in Computer Science., Springer (2006) 200–214
22. Mens, K., Kellens, A., Pluquet, F., Wuyts, R.: The intensional view environment. International Conference on Software Maintenance (ICSM) Industrial and Tool Volume (2005) 81–84
23. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design patterns: elements of reusable object-oriented software. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1995)
24. Wuyts, R.: A Logic Meta-Programming Approach to Support the Co-Evolution of Object-Oriented Design and Implementation. PhD thesis, Vrije Universiteit Brussel (2001)
25. Curino, C.A., Moon, H.J., Tanca, L., Zaniolo, C.: Schema evolution in wikipedia: toward a web information system benchmark. In Cordeiro, J., Filipe, J., eds.: International Conference on Enterprise Information Systems (ICEIS). (2008) 323–332
26. Hainaut, J.L.: A generic entity-relationship model. In: Proceedings of the IFIP WG 8.1 Conference on Information System Concepts: an in-depth analysis, North-Holland (1989)
27. Heering, J., Hendriks, P.R.H., Klint, P., Rekers, J.: The syntax definition formalism sdf—reference manual—. SIGPLAN Not. **24**(11) (1989) 43–75
28. van den Brand, M., van Deursen, A., Heering, J., de Jong, H., de Jonge, M., Kuipers, T., Klint, P., Moonen, L., Olivier, P., Scheerder, J., Vinju, J., Visser, E., Visser, J.: The ASF+SDF Meta-Environment: a Component-Based Language Development Environment. In Wilhelm, R., ed.: Compiler Construction (CC '01). Volume 2027 of Lecture Notes in Computer Science., Springer-Verlag (2001) 365–370
29. Hnatkowska, B., Huzar, Z., Kuzniarz, L., Tuzinkiewicz, L.: A systematic approach to consistency within UML based software development process. In Kuzniarz, L., Reggio, G., Sourrouille, J.L., Huzar, Z., eds.: Blekinge Institute of Technology, Research Report 2002:06. UML 2002, Model Engineering, Concepts and Tools. Workshop on Consistency Problems in UML-based Software Development. Workshop Materials, Department of Software Engineering and Computer Science, Blekinge Institute of Technology (2002) 16–29
30. Harel, D., Rumpe, B.: Modeling languages: Syntax, semantics and all that stuff, part i: The basic stuff. Technical report, Jerusalem, Israel, Israel (2000)
31. Lange, C., Chaudron, M.R.V., Muskens, J., Somers, L.J., Dortmans., H.: An empirical investigation in quantifying inconsistency and incompleteness of uml designs. In Kuzniarz, L., Huzar, Z., Reggio, G., Sourrouille, J.L., Staron, M., eds.: Proceedings of the IEEE Workshop on Consistency Problems in UML-Based Software Development II. (2003) 26–34
32. Finkelstein, A., Gabbay, D.M., Hunter, A., Kramer, J., Nuseibeh, B.: Inconsistency handling in multi-perspective specifications. In Sommerville, I., Paul, M., eds.: Proceedings of fourth European Software Engineering Conference (ESEC1993). Volume 717 of Lecture Notes in Computer Science., Springer (September 1993) 84–99 Garmisch-Partenkirchen, Germany.

33. Nentwich, C., Emmerich, W., Finkelstein, A., Ellmer, E.: Flexible consistency checking. ACM Trans. Softw. Eng. Methodol. **12**(1) (2003) 28–63
34. Küster, J.M.: Consistency Management of Object-Oriented Behavioral Models. PhD thesis, University of Paderborn (March 2004) Paderborn, Germany.
35. Nuseibeh, B., Easterbrook, S.M.: The process of inconsistency management: A framework for understanding. In: DEXA Workshop. (1999) 364–368