Composing Models for Detecting Inconsistencies: A Requirements Engineering Perspective

Gilles Perrouin[1], Erwan Brottier[2], Benoit Baudry[1], and Yves Le Traon[3]

[1] Triskell Team IRISA/INRIA Rennes Campus de Beaulieu, 35042 Rennes, France
[2] France Télécom R&D, 2 av. Pierre Marzin, 22 307 Lannion Cedex, France
[3] ENST Bretagne, 2 rue de la Châtaigneraie, CS 17607, 35576 Cesson Sévigné Cedex France
[1]{gperroui,bbaudry}@irisa.fr, [2]erwan.brottier@orange-ftgroup.com,
[3]Yves.letraon@telecom-bretagne.eu

Abstract. [**Context and motivation**] Ever-growing systems' complexity and novel requirements engineering approaches such as reuse or globalization imply that requirements are produced by different stakeholders and written in possibly different languages. [**Question/ problem**] In this context, checking consistency so that requirements specifications are amenable to formal analysis is a challenge. Current techniques either fail to consider the requirement set as a whole, missing certain inconsistency types or are unable to take heterogeneous (i.e. expressed in different languages) specifications into account. [**Principal ideas/ results**] We propose to use model composition to address this problem in a staged approach. First, heterogeneous requirements are translated in model fragments which are instances of a common metamodel. Then, these fragments are merged in one unique model. On such a model inconsistencies such as under-specifications can be incrementally detected and formal analysis is made possible. Our approach is fully supported by our model composition framework. [**Contribution**] We propose model composition as means to address flexibility needs in requirements integration. Threats to validity such as the impact of new requirements languages needs to be addressed in future work.

Keywords: model-driven requirements engineering, flexible inconsistency management, model composition.

1 Introduction

Cheng and Atlee [1] have reviewed state of the art of current requirements engineering research and identified future directions. Amongst those, two seem particularly relevant to address ever-growing system complexity, shorten engineering time and maximize value: *Globalization* and *Reuse*. Globalization suggest to engineer systems in geographically distributed teams in order to benefit from a continuous working force (24h/day), close distance of customers and resource optimization. Reuse offers to capitalize on requirements value by wisely

re-applying the same requirements in a product-line context. These promising research directions have a strong impact on the definition of the requirements themselves. First, requirements of a single system will be handled by several engineering teams having different habits and therefore inducing communication challenges [1]. Second, reusing requirements in a new context may imply having to deal with different formalisms used for their description. As mentioned by Sommerville [2], a Software Requirements Specification (SRS) is captured by a collection of viewpoints, described by system authorities (stakeholders, existing system documentation, and so on). Viewpoints encapsulate partial requirements information, described by heterogeneous models i.e. expressed in various languages (depending on stakeholders preferences and skills) and relating to different crosscutting concerns [3].

Globalization and reuse also represent a challenge for consistency management. Indeed, models forming viewpoints are likely to be inconsistent due to the amount of heterogeneous information involved and the number of stakeholders responsible of their productions. Requirements analysts need to detect inconsistencies among these models to reveal conceptual disagreements and drive the requirements elicitation process [4]. To do so, they need a global view of inconsistencies in order to decide whether they will tolerate inconsistency presence or not [5]. Model comparison techniques [4, 6–8] have been proposed to detect logical contradictions with respect to consistency rules. These techniques are relevant to find static and inter-model inconsistencies. But they have some important limitations. First, consistency rules in a multi-formalism context must be written for each possible pair of languages. Second, these techniques are inefficient to provide a measure of the overall SRS consistency since consistency rules checks models two by two. As a result, they are not suitable to detect under-specifications (a lack of information detected between two models may be resolved by a third one). Moreover, dynamic inconsistencies are undetectable because formal techniques enabling their detections can not be used as they require a global model of the SRS.

Composing models can help to overcome these limitations by providing one global model from a set of models providing an unified view [9] of the requirements with respect to a particular purpose (e.g. functional requirement simulation). Regarding inconsistencies detection, model composition translates the inter-model consistency problem into an intra-model consistency one. This has numerous advantages. First, consistency rules can be defined on one unique metamodel and hence are much easier to specify. Second, dynamic inconsistencies can be readily checked by formal tools.

However, current composition techniques [9–14] do not fully address our problem. Indeed, they do not support the composition of heterogeneous models since they compose models in the same formalism. Most of these approaches assume that models are conforming to their metamodel prior to their composition which is not common place in requirements engineering practice [15]. Finally, they do not emphasize of ensuring traceability during composition which is required to determine inconsistency source, such as stakeholders conflicts.

Building on our experience on model composition [13], we propose in this paper a generic composition process addressing the above issues. First, we extract information from heterogeneous models and translate it in terms of a set of model fragments. This step is called interpretation. The second step, called fusion, builds a global model by composing model fragments. The resulting global model can be analyzed with respect to under-specifications and other inconsistency types through dedicated diagnostic rules potentially covering the SRS as a whole. By postponing inconsistency detection in the global model it is possible to resolve inconsistencies in an order only determined by the requirements engineers. Therefore, we provide them the flexibility required to drive inconsistency management. We also automatically compute traceability links between elements of input models and the global one. This is useful to trace inconsistencies back in models, providing valuable feedback to stakeholders. This process is fully supported by a model-driven platform [13, 16], integrated into the Eclipse framework. Section 2 outlines our model composition process. Section 3 details the fusion step and illustrates how inconsistent models can be composed. Section 4 illustrates how various kinds of inconsistencies can be detected. Section 5 highlights some relevant work. Section 6 concludes the paper and sketches some interesting perspectives.

2   Process overview

In this section, we describe our composition-and-check process. It is a result of researches carried out in the context of the R2A[4] project (R2A stands for Requirement To Analysis) project, initiated in collaboration with THALES and FRANCE TELECOM. Its goal is the definition of a framework for analyzing requirements, simulating functional requirements [16] and generating software artifacts from them [17]. This process is completely based on a model-driven approach and consists of three sequential steps, as shown in Figure 1. It starts from a set of input models, expressed in various *input requirements languages* (IRL). These models are composed during the *interpretation* and *fusion* steps, presented in section 2.1. These steps result in a *global model* and a *traceability model*. These two models are the inputs of a *static analysis* step which consists in checking the global model according to consistency rules and producing a consistency verdict. This last step is described in section 2.2.

2.1   Composition

Input models to compose are produced by stakeholders. They capture specific parts of the software requirements specification (SRS) and describe partial information of the requirements. They may be inconsistent and can be described

---

[4] `http://www.irisa.fr/triskell/Softwares/protos/r2a/r2a_core?set_language=en`
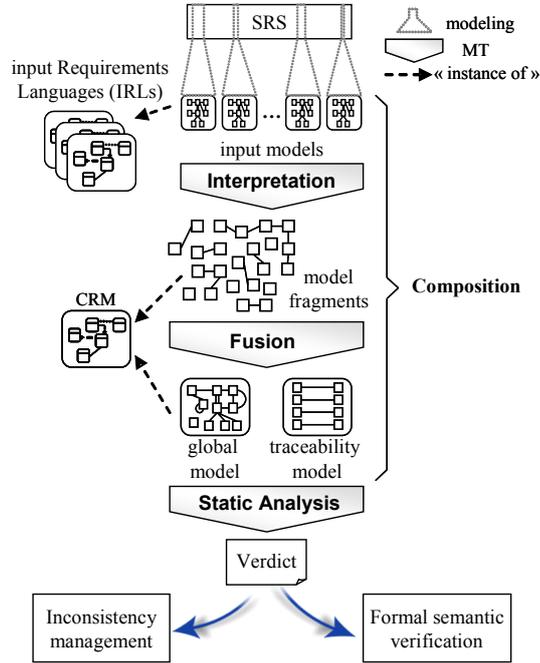
Fig. 1. Overview of the composition-and-check process

with various input requirements languages as depicted in Figure 1 (a preliminary parsing step presented in [13] is required for textual specifications.). Furthermore, they describe different concerns of the system-to-be. Composing such input models requires to state precisely the following: 1) Which information must be extracted from input models, 2) How this information is obtained from these input models and 3) How extracted pieces of information must be combined to obtain one global model.

Core Requirements Metamodel. The *Core Requirements Metamodel* (CRM) defines information that will be extracted from input models and composed. Two main factors influence the definition of the CRM. The first one is related to the answer that we will give to the first point i.e. the elicitation of a subset of the IRLs' concepts on which we will base analysis on the global model. This elicitation is the result of a negotiation between stakeholders to determine what are the most important concepts according to their respective viewpoints.
The second important factor is the type of analysis that is targeted by the check process. If dynamic inconsistency checking is required, a formal operational semantics of the CRM has to be given.

Interpretation. The first step of the process, called *interpretation*, addresses the second point. This step is detailed in previous work [13] and is beyond the scope of this paper. Basically, the interpretation extracts relevant information in input models and translates it in terms of *model fragments*, which are instances of the CRM. The interpretation is governed by a set of interpretation rules matching IRLs' concepts (with respect to optional guards) and producing corresponding model fragments instances of the CRM. These interpretation rules have been defined in collaboration with THALES' requirements analysts in order to validate "correctness by construction" of interpretation rules.

Fusion. The second step of the process called *fusion* addresses the third point. From a model-driven perspective, the fusion step is supported via a model composition technique which provides a high-level of flexiblity (see Section 3.3 for details). As for interpretation, fusion is described by a set of *fusion rules*. The fusion consists in detecting and resolving *overlaps* between model fragments. An overlap is a situation where two sets of related model elements in different models are semantically equivalent i.e. designate common features of the domain of discourse [18]. Overlap resolution aims at producing a compact representation of information captured by interpretation, i.e. the global model. Without replacing a complete semantic analysis, the fusion makes explicit semantic links between input models. This step is detailed and illustrated in section 3.
Interpretation and fusion rules automatically build traceability information necessary to identify elements which are the cause of an inconsistency.

2.2 Static Analysis

The third step of the process, called static analysis, is performed once the global model is available. Two kinds of consistency rules are checked. The first ones, called structural inconsistencies, check if the global model fulfills at least the CRM constraints expressed with MOF (cardinalities, composition...). Two kinds of inconsistencies can be detected at this stage:

– *Under-specification* is detected if one property value of one object has fewer elements than specified by the property cardinality. It means that information is missing in input models and the global model is incomplete with regards to the targeted formal analysis technique,
– *Logical contradiction* is detected if one property value of one object has more elements than specified by the property cardinality. It means that at least two input models overlap semantically but this overlap is inconsistent.

The second kind of consistency rules, called *static semantics inconsistencies*, is complex and is generally described with OCL rules. Intuitively, these rules correspond to well-formedness rules defining business-specific restrictions on metamodels. However these rules can be difficult to write in OCL for stakeholders who do not have a technical expertise. To help such stakeholders, we propose the notion of *diagnostic rules* which are easier to write for stakeholders

and enable to provide meaningful information in a non-technical form when a rule is violated. The composition-and-check process can be used iteratively in order to limit the amount of managed information and the number of inconsistencies to solve. As the fusion step can take as an input the global model (it is just a large model fragment). It is then possible to check how a new input model impacts a consistent global model. When no inconsistencies are detected, the targeted formal verification technique can be applied. When input models (and corresponding parts of the SRS) are updated to take into account results of the formal verification, a new cycle can be started. These points will be detailed in section 4.2.

3 Combining inconsistent models via Fusion

### 3.1 Running Example: the RM

The RM metamodel is the current CRM of the R2A platform. It captures a functional and a data description of the system. It is meant to capture functional requirements and the control flow between them. RM also allows for requirements simulation and system test cases generation within R2A platform [17]. Figure 2 illustrates an excerpt of the RM metamodel. It describes a state-based formalism where system actions are described as use cases (metaclass UseCase), enhanced with their activation conditions and effects (relationships PreCondition and PostCondition expressed as first order logic expressions).

Only few concepts defining these expressions are showed in Figure 2 (the Expression subclasses). Expressions contain references to property values of the system, represented by the metaclass PropertyValue. The OCL constraint 1 in Figure 2 ensures that referenced property values in any expression exist in the model (constraint 2 checks their type conformance). Use case contains a set of formal parameters (metaclass Parameter). Each parameter represents an Entity involved in the use case, which plays a particular Role (for instance "actor"). Only the actor parameter of a use case can trigger it and a use case has only one actor, as expressed by the OCL constraint 3.

Other notions in Figure 2 describe basically an entity-relationship diagram. Entities are either business concepts (Concept) or actors (Actor) of the system (entity which can trigger at least one use case). Entities have properties which represent their possible states (Attribute) and relationship with others (Relation). Properties have a type (DataType) which can be a Boolean-Type or an EnumerationType (a finite set of literal values, representing strings, integers or intervals). Instances of attributes and relations are property values of the system. A system state is a set of entity instances and property values, corresponding to a system configuration at a given moment. The reader can refer to [13, 15] for more details on this metamodel.

3.2 Dealing with Inconsistent Fragments

Figure 3 presents a set of RM model fragments, obtained by interpreting a set of input models (interpretation step). These input models represent the specifi-

Fig. 2. A part of the RM metamodel used in R2A platform
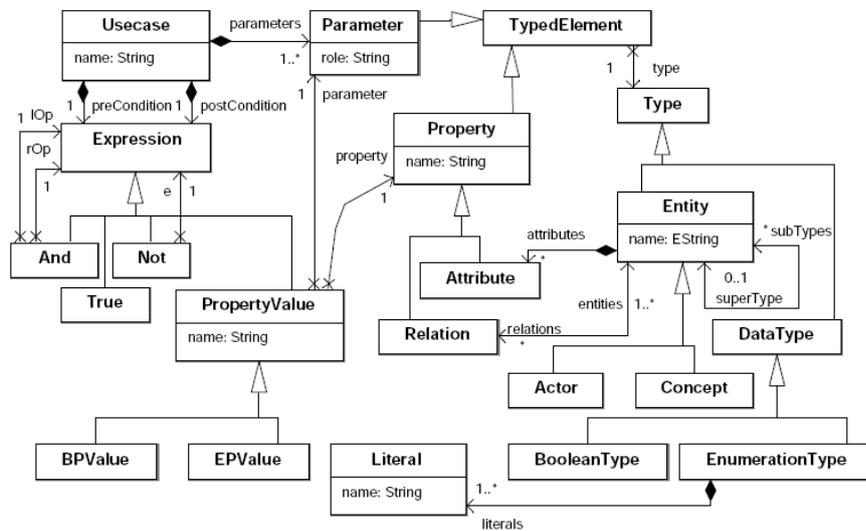
(1) context PropertyValue : let x = parameter.type.oclAsType(Entity) in property = x.attributes.union(x.relations)
    ->select(p|p.name = name)->one
(2) context PropertyValue : self.oclTypeOf(BPValue) and property.type.oclTypeOf(BooleanType) or
    self.oclTypeOf(EPValue) and property.type.oclTypeOf(EnumerationType)
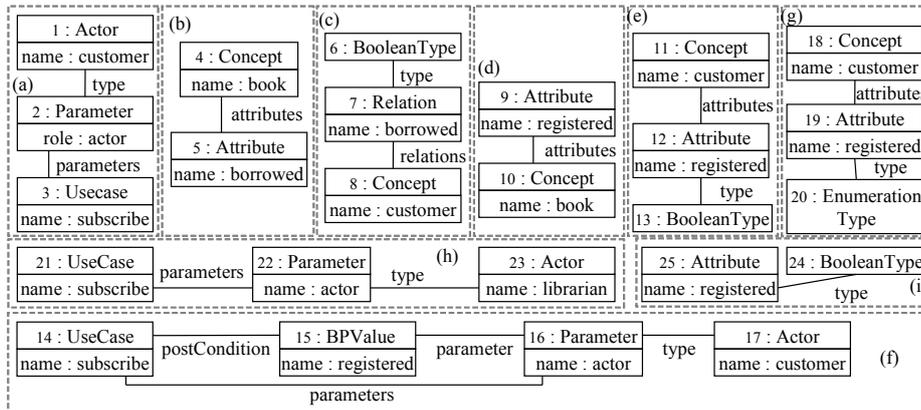(3) Context Usecase : parameters.select(p|p.role="actor").size() = 1

Fig. 3. Examples of RM model fragments

cation of a library management system (see [13] for details). Input models are written in RDL (Requirement Description Language [13]) which is a constrained form of natural English. For example, one sentence of library management system, interpreted in terms of model fragments of Figure 3 is:

```
The ''book'' must be registered before the ''customer'' can
''borrow'' the ''book''
```

Each fragment handles a piece of information, extracted from one input model by one interpretation rule as explained in section 2.1. The fragment (a) declares a use case subscribe which can be triggered by an actor *customer*. The fragment (b) states the existence of a business concept *book* which has a state *borrowed*, without information on its type. To handle such inconsistent fragments in a model (prior and after their fusion), it is necessary to allow non-compliance with respect to basic well-formedness rules of the metamodel such as multiplicities or types. We introduce the notion of a *relaxed metamodel* to cope with this situation.

Definition 1. *A relaxed CRM is a metamodel where the following properties hold:*

– *All metaclasses are considered as concrete (i.e. instantiable),*
– *All multiplicities between metaclasses are considered as '*',*
– *There is no containment relationship (treated as regular associations).*

Thanks to this notion it is possible to interpret partial and inconsistent model elements and to combine them. As mentioned in Section 2, inconsistency checking is progressively performed on the global model. Finally the global model is made consistent with respect to the original (un-relaxed) CRM. This process will be detailed in Section 4.

3.3 Fusion principles

One simple approach to perform model fusion is to compare objects two by two and combine them when with respect to syntactical equivalence. Yet, this raises two issues (i) *designation clashes* and (ii) *type clash*. Designation clash [19] occurs when a single syntactic name in the requirements specification designates different real-world concepts (e.g. the pair (9,12) refers to different concepts while having the same name). A type clash arises when two different types ar given for the same concept (e.g. the pair (1, 8) refers to a customer in two different ways). It can be seen as specialization of a *terminology clash* [19]. To alleviate these issues, we need to let requirements analysts define fine-grained rules specializing fusion and resolve these clashes. To this aim, we propose two kinds of fusion rules (FR): *equivalence rules* (ER) and *normalization rules* (NR).
Rules of the first type define *equivalence ranges* and describe how to resolve them. An equivalence range is a set of objects which are considered as equivalent. Resolution targets the replacement of equivalent objects by one new object, where

properties have been set properly to keep the whole information. Normalization rules aim at transforming the model after ER executions so that a violation of conformity reflects an inconsistency. Figures 4 and 5 give examples of FR specifications for the RM metamodel presented in Figure 2.

$$
\begin{aligned}
&\textbf{ER1} \quad (a_1, a_2 : \text{ATTRIBUTE}): \text{ATTRIBUTE} \\
&\quad\text{-}\quad a_1.name = a_2.name \\
&\qquad\qquad \wedge\ a_1.attributes^{-1} = a_2.attributes^{-1}. \\
&\textbf{ER2} \quad (r_1, r_2 : \text{RELATION}): \text{RELATION} \\
&\quad\text{-}\quad r_1.name = r_2.name. \\
&\textbf{ER3} \quad (r : \text{RELATION}, a : \text{ATTRIBUTE}): \text{RELATION} \\
&\quad\text{-}\quad a.name = r.name. \\
&\quad\text{-}\quad linkedEntities = \text{equRanges.collect}(r : \text{RELATION} \mid r.\text{linkedEntities}) \cup \text{c.collect}(a : \\
&\qquad\qquad \text{ATTRIBUTE} \mid a.\text{owningEntity}). \\
&\textbf{ER4} \quad (u1, u2: \text{USECASE}): \text{USECASE} \\
&\quad\text{-}\quad u1.name = u2.name. \\
&\textbf{ER5} \quad (p_1, p_2 : \text{PARAMETER}): \text{PARAMETER}. \\
&\quad\text{-}\quad p_1.name = p_2.name. \\
&\textbf{ER6} \quad (a_1, a_2 : \text{ACTOR}): \text{ACTOR}. \\
&\quad\text{-}\quad a_1.name = a_2.name. \\
&\textbf{ER7} \quad (c : \text{CONCEPT}, a : \text{ACTOR}): \text{ACTOR} \\
&\quad\text{-}\quad c.name = a.name. \\
&\textbf{ER8} \quad (c_1, c_2 : \text{CONCEPT}): \text{CONCEPT}. \\
&\quad\text{-}\quad c_1.name = c_2.name. \\
&\textbf{ER9} \quad (b_1, b_2 : \text{BOOLEANTYPE}): \text{BOOLEANTYPE}. \\
&\quad\text{-}\quad b_1.type^{-1} = b_2.type^{-1}.
\end{aligned}
$$

Fig. 4. Examples of equivalence rules

An ER is defined as an equivalence range constructor and an equivalence range resolution (constructor and resolution in the remainder). The constructor is a boolean expression which takes as inputs a pair of objects and returns true if they are equivalent. It aims at defining a set of equivalence ranges in the context of its owning rule. The resolution is used to replace all objects in the equivalence ranges by a new object which captures their semantics. ERs are expressed with three ordered elements, listed in Figure 4: a *signature* (first line), a *constructor* boolean expression (the first element of the list) and a set of *resolution directives* (the second element being optional and used only in ER3). The signature handles the name of the rule, a type filter on the pair of objects checked by the constructor and the type of the object created by the resolution (*return type*). ER1 specifies for instance that objects of type ATTRIBUTE which have the same name and are related to the same ENTITY (*attributes*$^{-1}$ points out the opposite reference of *attributes*) are part of the same equivalence range. This ER illustrates how context can be part of overlaps identification. The resolution of such an equivalence range will produce an instance of ATTRIBUTE, as specified by its return type.

```
NR1 : Execution of an imperative form of the constraint 1 in Figure 2 for each
      PROPERTYVALUE.
NR2 : Creation of an And tree with elements of u.preCondition for each USECASE.
NR3 (uc : USECASE) ? : card(uc.preCondition) = 0;
        ! : uc.preCondition := TRUE.new;
```

Fig. 5. Examples of normalization rules

A resolution directive describes how to set the value of a particular property for the object created by the resolution. If no resolution directive is defined, the default policy is applied. As defined by Sabetzadeh et al. [9], it consists in making the union of the property value of each object in the equivalence range (with basic equality resolution for primitive types such as string or integer). Yet, some property values cannot be resolved by union (for instance the kind of a UML class representing one abstract class and one interface). In such cases, resolution directives are useful to resolve these overlaps. As an example, each property value of ATTRIBUTE instances created by the ER4 resolution is evaluated by the default policy as opposite to the *linkedEntities* property values of RELATION instances created by the ER3 resolution. The fusion of model fragments may not be performed in one step (identification of all equivalence ranges and resolution). Indeed, an object can be contained by more than one equivalence range. As an example, the object 1 is also contained by (1, 17) ER6. The stopping criterion of the fusion is satisfied when no more equivalence ranges have been identified. Some inconsistencies in a model do not reveal inconsistencies in the information captured but only irrelevant ways to express this information. For instance, an instance of Usecase in a RM model must have one pre-condition exactly. If this use case has no condition of activation, its pre-condition must be an instance of the metaclass TRUE. This mapping to TRUE must be done if no partial specifications describe information about this pre-condition. NR3 is specified for this purpose.

The main part of the fusion algorithm is given in Figure 7. It processes a set of objects (from model fragments to merge) and a set of ER. It iterates until no more equivalence ranges have been resolved (or identified). The loop contains two steps: equivalence range identifications (lines 04-07) and their resolutions (lines 08-13). The method elicit(l:list of objects) removes all objects passed as parameter (it also deletes links between them and remaining objects). It is used to remove objects contained by a resolved ER. Figure 6 gives the final model obtained by executing this algorithm on model fragments of Figure 3 according to fusion rules of Figure 4 and 5. It is an instance of the relaxed CRM.

3.4 Traceability computation

The composition process generates automatically a traceability model as introduced in section 2.1. This model stores the history of all kinds of rules (interpretation, fusion and normalization) that have been executed during the com-
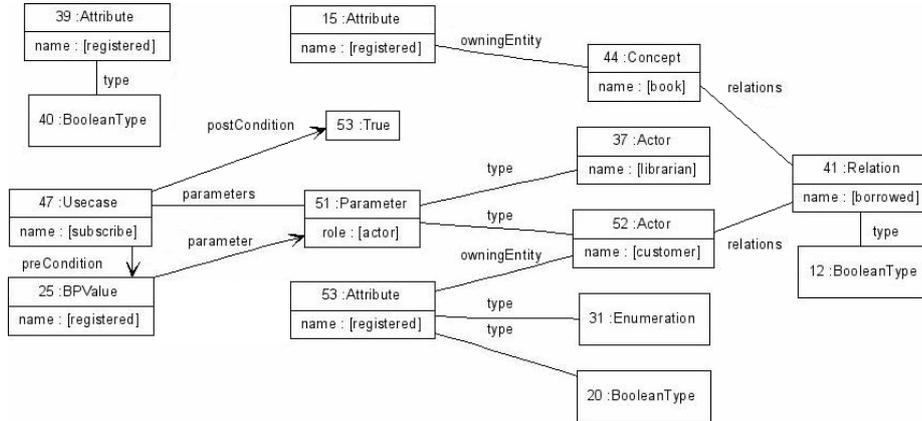
Fig. 6. Result of the application of Fusion rules on model fragments (Figure 3)

```
0       Algo Fusion(l1 : List<Object>, l2 : List<ER>) is
1       var resolved : List
2       do
3        resolved.clear()
4        forAll o1 in l1
5         forAll o2 in l1
6          forAll er in l2
7           if (er.constructor(o1, o2)) then er.equivalences.add(o1, o2)
8         forAll er in l2
9          forAll r in f.equivalenceRanges
10          var o : Object := er.resolution(r)
11          l1.add(o)
12          resolved.addAll(eq)
13        l1.elicit(resolved)
14       until(resolved.isVoid())
```

Fig. 7. Fusion Algorithm (equivalence rules application)

position process. Each rule execution is associated to model elements that have been matched and produced. Model elements pertain either to input models, interpreted and composed model fragments or to the global model. Given such a traceability model, it is possible to compute a connected graph where nodes are model elements involved in at least one rule and vertices relates elements matched and produced for each rule.

4 Inconsistency detection

We illustrate in this section our inconsistency detection process, introduced in section 2.2. This process is composed of two activities performed in parallel: *structural inconsistency detection* and *static semantics inconsistency detection.*

4.1 Structural Inconsistency Detection

Structural inconsistency detection checks conformance of the model with respect to the CRM definition expressible through MOF. For example, one constraint on the RM metamodel requires that any UseCase has at least one Parameter. Structural inconsistencies comprise logical contradictions: attribute *registered* of customer (object 53 in Figure 6) has two types which is clearly a non-sense. Concerning under-specifications, attribute *registered* (object 15) of *book* has no type at all. Structural inconsistencies for a given CRM are automatically detected by MOF-compliant tools. Therefore there is no need to define them for a new CRM. When no more structural inconsistency is detected, the global model can be cast to an instance of the original (un-relaxed) CRM.

4.2 Static Semantics Inconsistency Detection

We distinguish two categories of static semantics inconsistencies depending on the stakeholders who specify them. *Well-formedness* inconsistencies are related to rules domain experts define on the CRM (e.g. constraint 1 for the RM metamodel in Figure 2). *Custom* inconsistencies are violation of rules defined by requirements analysts. Enforcing such rules may be required to enable further formal analysis on the model. As mentioned in section 3, fusion outputs the global model as an instance of the relaxed CRM. Since for flexibility reasons [5] we do not want to impose a sequence in inconsistency detection activities, we cannot assume that the global model can be cast to an instance of the CRM. We therefore need to define well-formedness and custom rules on the relaxed CRM. While OCL is often used to specify static semantics rules, defining navigation paths across scattered elements can be tricky and rule violation feedback useless for stakeholders.

We offer the possibility to requirements analysts to define diagnostics rules (DR), defined for a given version of the relaxed CRM, to solve these problems. Diagnostics rules encapsulate a guard and a textual template. The guard is based on pattern matching as for interpretation and fusion rules. Hence, it is no longer required to specify navigation through the model to retrieve model elements. The template is a natural language sentence parameterized with expressions referring matched elements and navigating the global model. When a DR matches, an inconsistency is detected, the template is instantiated and added to a verdict log which can be reviewed by non-experts. Traceability links can be then navigated until pointing out inconsistent requirements in input models and rules involved in the production of elements responsible of the inconsistency. A few examples based on the RM metamodel are provided in Figure 8a. For instance,
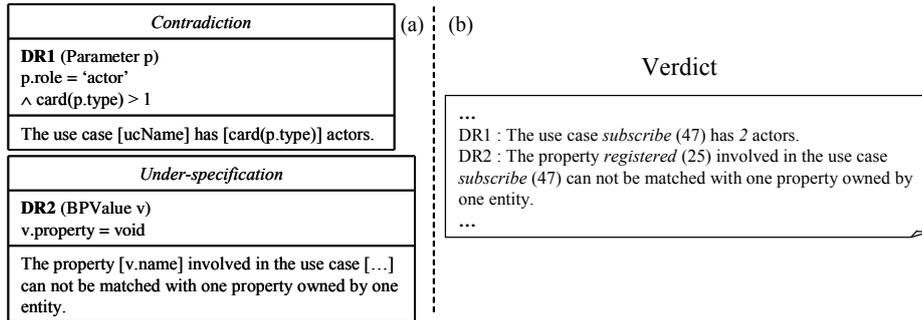
Contradiction	(a) ┊ (b)
DR1 (Parameter p) p.role = 'actor' ∧ card(p.type) > 1	
The use case [ucName] has [card(p.type)] actors.	

Under-specification
DR2 (BPValue v) v.property = void
The property [v.name] involved in the use case [...] can not be matched with one property owned by one entity.

Verdict

...
DR1 : The use case *subscribe* (47) has *2* actors.
DR2 : The property *registered* (25) involved in the use case *subscribe* (47) can not be matched with one property owned by one entity.
...

Fig. 8. Some DRs for the metamodel RM and an excerpt of the verdict

DR1 declares that there is a contradiction if a use case has more than one actor. Indeed, this DR is related to the OCL constraint 3 defined Figure 2). As the verdict illustrates (Figure 8b), the feedback for this rule is much more understandable for non-experts.

When both static semantics and structural inconsistencies are resolved we cast the global model to an instance of the original CRM which is fully consistent and amenable to further analysis.

While having no inconsistency detection sequence is a good point for flexibility, it can be disturbing for requirements analysts to figure out which rules have to be checked first, especially if the CRM is complex and there are numerous rules. In such cases we propose to manage inconsistency resolution incrementally by stepwise resolving groups of diagnostics rules. Freed from restriction such as checking of structural inconsistencies prior to static semantics ones, domain experts can drive inconsistency management in accordance with conceptual and methodological grounds rather than technical limitations.

5   Related Work

Zave and Jackson defined in [12] the scientific foundations of multi-formalism composition. They describe a formal CRM designed for formalizing a wide range of metamodels where composition is the conjunction of elements extracted from models. They describe a notion of functions dedicated to each input language for assigning input model semantics in the CRM. They then discuss a few methodological points on how to perform inconsistency checking. However, they do not discuss inconsistency types that can be verified and do not offer any implementation supporting their vision. More concrete composition solutions have been proposed as a way to perform semantic analysis [10, 11]. In [10], the authors defines a framework to produce a HOL (High Order Logic) model from models expressed in different notations and analyze the composed model with HOL tools. Ainsworth et al. [11] propose a method for composing models written in

Z. Relations between models are described with functional invariants and some Z proof obligations can be performed to verify composition correctness. However, these two approaches require that a significant amount of work shall be done manually, either for the pre-processing of Z models [11] or relevant information must be extracted and translated into a HOL specification by hand in [10]. Moreover, they do not process inconsistent models.

Kompose [14] is a meta-modeling approach built on the Kermeta language. It targets the automatic structural composition of aspects in software development processes. It is a rule-based approach where equivalence between objects is automatically calculated with regards to object structure. Kompose assumes that homogeneous input models (i.e. instances of a unique metamodel) are structurally consistent prior to their composition. Kolovos et al [20] propose a model merging language able to compose heterogeneous models which uses pattern-matching to merge models. However, similarly to compose they need to avoid conflicts before merge which restricts the inconsistency types that can be fixed and the global model and limit flexibility with respect to inconsistency management. Sabetzadeh and Easterbrook [9] provide an interesting composition framework based on a formal definition of models to compose. The composition operator (category-theoretic concept of colimit) is formally defined and traceability links are automatically inferred. However this operator requires the model type of [21] which restricts highly the accepted IRLs. As opposed to Kompose and our approach, equivalence must be given by hand by the requirements analyst and composition only works for homogeneous models.

6   Conclusion

Dealing with inconsistencies across multiple models is a critical issue that requirements analysts and software engineers have to face nowadays. In this paper, we have shown how, by translating the problem from managing inconsistencies amongst heterogeneous models to managing inconsistencies within a single model, stakeholders' task can be greatly facilitated. In particular, we proposed a novel model composition mechanism which able to compose partial and possibly inconsistent models. Hence, various categories of inconsistencies can be checked on the resulting global model. Furthermore, as the order of inconsistencies to be solved is not prescribed by the approach, requirements analysts can flexibly drive the inconsistency management depending on the context. Native traceability supported by our implementation enables to report inconsistencies on the original models thus easing the determination of inconsistency causes. We are currently working on integrating our platform with formal analysis tools to obtain a complete requirements validation chain. Integration is performed by means of a model transformation translating CRM instances into models analyzable by the targeted tool.

In the future, we would like to acquire experience on the *adaptability* of the approach to various contexts and input languages. In particular, we will assess

the impact of the introduction of a new input language on fusion rules and on the CRM.

References

1. Cheng, B.H.C., Atlee, J.M.: Research Directions in Requirements Engineering. In: FOSE at ICSE, Washington, DC, USA, IEEE Computer Society (2007) 285–303
2. Sommerville, G.K., Ian: Requirements Engineering with Viewpoints. Software Engineering Journal (1996)
3. Rashid, A., Moreira, A., Araújo, J.: Modularisation and composition of aspectual requirements. In: AOSD'03, Boston, Massachusetts, USA (2003) 11 – 20
4. Easterbrook, S., Nuseibeh, B.: Using viewpoints for inconsistency management. Software Engineering Journal **11**(1) (1996) 31–43
5. Nuseibeh, B., Easterbrook, S., Russo, A.: Making inconsistency respectable in software development. Journal of Systems and Software **58**(2) (2001) 171–180
6. Nuseibeh, B., Kramer, J., Finkelstein, A.: A framework for expressing the relationships between multiple views in requirements specification. IEEE TSE **20**(10) (1994) 760–773
7. Nentwich, C., Emmerich, W., Finkelstein, A.: Flexible consistency checking. ACM TOSEM (2001)
8. Kolovos, D., Paige, R., Polack, F.: Detecting and Repairing Inconsistencies across Heterogeneous Models. In: ICST, Los Alamitos, CA, USA, IEEE Computer Society (2008) 356–364
9. Sabetzadeh, M., Easterbrook, S.: An algebraic framework for merging incomplete and inconsistent views. In: RE'2005, IEEE (Aug.-2 Sept. 2005) 306–315
10. Day, N., Joyce, J.: A framework for multi-notation requirements specification and analysis. 4th ICRE (2000) 39–48
11. Ainsworth, M., Cruickshank, A., Groves, L., Wallis, P.: Viewpoint specification and Z. Information and Software Technology **36**(1) (1994) 43–51
12. Zave, P., Jackson, M.: Conjunction as Composition. ACM TOSEM **2**(4) (1993) 379–411
13. Brottier, E., Baudry, B., Traon, Y.L., Touzet, D., Nicolas, B.: Producing a Global Requirement Model from Multiple Requirement Specifications. In: EDOC. (2007) 390–404
14. France, R., Fleurey, F., Reddy, R., Baudry, B., Ghosh, S.: Providing Support for Model Composition in Metamodels. In: EDOC, Annapolis, MD, USA (2007)
15. van Lamsweerde, A., Letier, E., Ponsard, C.: Leaving Inconsistency. In: ICSE workshop on "Living with Inconsistency". (1997)
16. Baudry, B., Nebut, C., Traon, Y.L.: Model-driven engineering for requirements analysis. In: EDOC. (2007) 459–466
17. Nebut, C., Fleurey, F., Le Traon, Y., Jézéquel, J.M.: Automatic test generation: A use case driven approach. IEEE TSE (2006)
18. Spanoudakis, G., Finkelstein, A.: Overlaps among requirements specifications. In: ICSE workshop on "Living with Inconsistency". (1997)
19. van Lamsweerde, A., Darimont, R., Letier, E.: Managing Conflicts in Goal-Driven Requirements Engineering. IEEE TSE **24**(11) (1998)
20. Kolovos, D.S., Paige, R.F., Polack, F.A.C.: Merging Models With the Epsilon Merging Language EML. In: MODELS, Springer LNCS 4199 (2006)
21. Corradini, A., Montanari, U., Rossi, F.: Graph processes. Fundamenta Informaticae **26**(3-4) (1996) 241–265