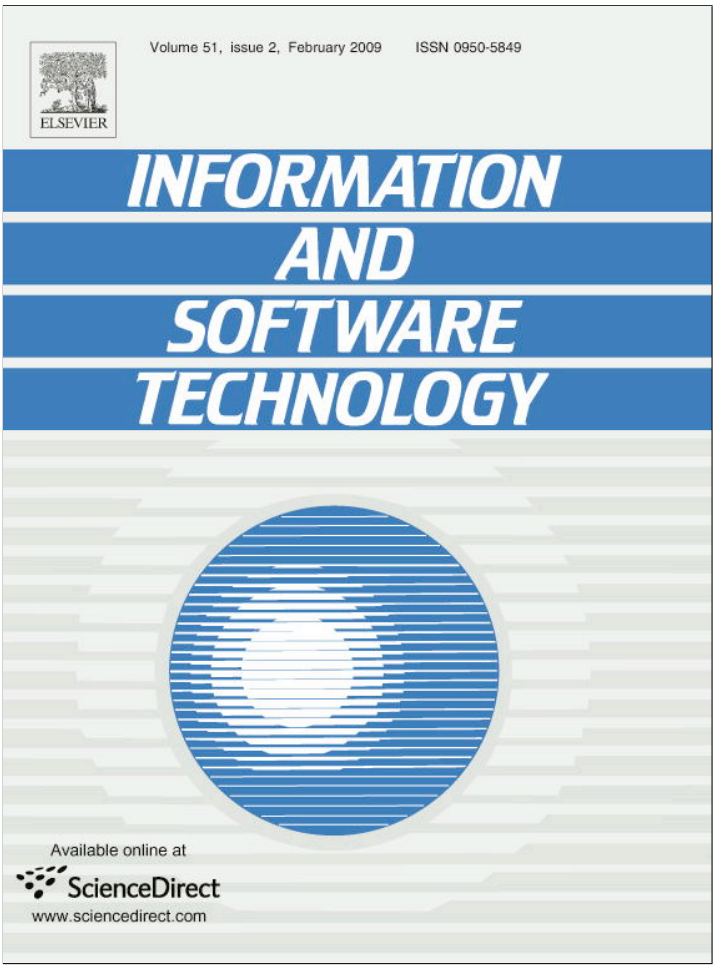

Information and Software Technology 51 (2009) 512–527

Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof

Automated traceability analysis for UML model refinements

Lionel C. Briand [b,*], Yvan Labiche [a], Tao Yue [a]

[a] Carleton University, Software Quality Engineering Laboratory, 1125 Colonel By Drive, Ottawa, Canada ON K1S 5B6
[b] Simula Research Laboratory, University of Oslo, P.O. Box 134, Lysaker, Norway

ARTICLE INFO

Article history:
Received 13 December 2007
Received in revised form 5 June 2008
Accepted 15 June 2008
Available online 22 July 2008

Keywords:
UML
OCL
Impact analysis
Traceability link
Refinement

ABSTRACT

During iterative, UML-based software development, various UML diagrams, modeling the same system at different levels of abstraction are developed. These models must remain consistent when changes are performed. In this context, we refine the notion of impact analysis and distinguish horizontal impact analysis–that focuses on changes and impacts at one level of abstraction–from vertical impact analysis–that focuses on changes at one level of abstraction and their impacts on another level. Vertical impact analysis requires that some traceability links be established between model elements at the two levels of abstraction. We propose a traceability analysis approach for UML 2.0 class diagrams which is based on a careful formalization of changes to those models, refinements which are composed of those changes, and traceability links corresponding to refinements. We show how actual refinements and corresponding traceability links are formalized using the OCL. Tool support and a case study are also described.

1. Introduction

The use of the Unified Model Language (UML) [37] for complex systems leads to a large number of inter-dependent UML diagrams that have to be consistent, e.g., the operations used in sequence diagrams must be defined in class diagrams. Furthermore, recent development methodologies, such as the Rational Unified Process [26], promote successive modeling iterations evolving and refining models until the final product is complete. As a result, for a specific system, different versions of the same UML diagrams are incrementally produced at different levels of abstraction. In the simplest case, one can consider two standard analysis and design abstraction levels [9]. An iterative development process should ensure that these two models remain consistent as they are incrementally refined and changed.

Using and updating these different models should be supported by some tool infrastructure, and one way to cope with the maintenance of such models is to comply with the Model Driven Architecture (MDA) framework [25]. According to MDA, a platform independent model (PIM)—the analysis model—is transformed into a platform specific model (PSM)—the design model, which is itself transformed into code. To allow (fully) automated transformations, MDA requires that tools "maintain the relationship between PIM and PSM, even when changes to the PSM are made. Changes in the PSM will thus be reflected in the PIM, and high-level documentation will remain consistent with the actual code." [25] In practice, when such transformations are not automated, an essential requirement is that some form of traceability between the models must be created and maintained and support must also be provided to facilitate the change of a PSM when its corresponding PIM changes.

Impact analysis is defined as "identifying the potential consequences of a change, or estimating what needs to be modified to accomplish a change." [4] In the context of UML-based iterative development, we refine this notion and consider horizontal impact analysis (HIA) and vertical impact analysis (VIA). HIA focuses on changes and impacts at one level of abstraction, and has been addressed by existing impact analysis work (e.g., [5,6]), whereas VIA focuses on changes at one level of abstraction and their impact at another level of abstraction. (This is similar to the notion of horizontal and vertical consistency between models [16].) This is illustrated in Fig. 1 where we consider two levels of abstraction. A change to a sequence diagram at the analysis (abstraction) level may impact other analysis diagrams: this is *horizontal* impact analysis. The same change may also impact model elements at the design (abstraction) level: this is *vertical* impact analysis.

Both HIA and VIA require some level of traceability. Traceability is "the ability to trace between software artifacts generated and modified during the software product life cycle." [4] In the case of HIA, traceability must exist between model elements at the same level of abstraction, and those links are established during horizontal evolution (Fig. 1), e.g., during analysis, an entity class receives new attributes, and traceability links are created between the class and the new attributes. Existing impact analysis techniques [6] have relied on such traceability links. In the case of VIA, traceability must exist between model elements at the more

* Corresponding author.
E-mail addresses: briand@simula.no (L.C. Briand), labiche@sce.carleton.ca (Y. Labiche), tyue@sce.carleton.ca (T. Yue).

0950-5849/$ - see front matter © 2008 Elsevier B.V. All rights reserved.
doi:10.1016/j.infsof.2008.06.002

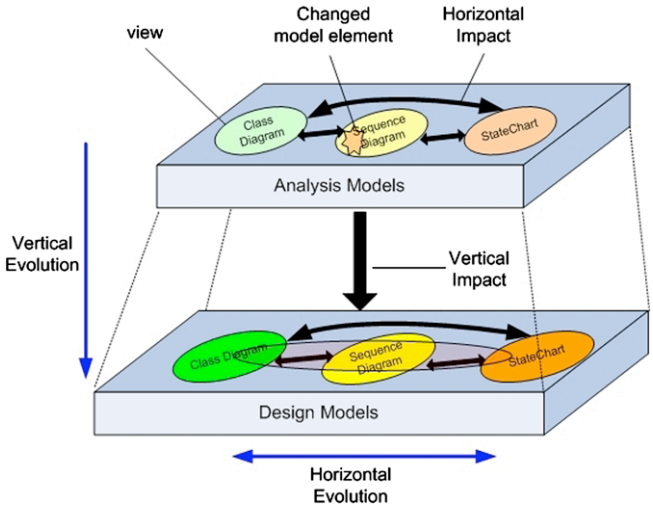

Fig. 1. Horizontal vs. vertical impact analysis.

abstract (analysis) level and model elements at the more refined (design) level, and traceability links are established during vertical evolution (Fig. 1). For example, the designer finds a state-dependent class in the analysis model and decides to use the state design pattern in the design (vertical evolution), in which case, links are created between the class in the analysis model and the pattern classes in the design model. In order to identify traceability links for supporting VIA, we must either capture (during vertical evolution) or determine (by comparison of models) the *intent* of the designer that lay behind changes when models are refined. Intent is modeled in this paper by a taxonomy of *refinements*. Rules associated with different types of refinements are then used to automatically identify refinements and establish traceability links.

A refinement is "used to model transformations from analysis to design and other such changes." [46]. A more general definition is given in [44], in which refinement is defined as 'a transformation that adds more detail to an existing model'. In [10, p. 215], the concept of refinement is defined as "a detailed description that conforms to another (its abstraction). Everything said about the abstraction holds, perhaps in a somewhat different form, in the refinement. Also called realization". The notion of refinement bears some similarities with the notion of refactoring (e.g., for UML diagrams [34]), which is defined as "a disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behaviors" in [17]. Refinements need to maintain the external behavior of a system as refactorings do; however performing refinements is to refine an abstract model into a detailed design model (i.e., vertical evolution) and performing refactorings aim at improving the internal structure of systems (horizontal evolution). In [19] the authors propose the concept of *change pattern*, which is defined as "a general repeatable solution to a commonly occurring problem in software evolution". The authors state that change patterns have no restriction on the preservation of models' behavior and are more general than refactoring. In this sense, the term change pattern has the same meaning as our concept of refinement. Applying design patterns to solve commonly occurring design or modeling problems can be thought of as a special type of refinement. In other words, our refinement notion does not distinguish the notions of refinement, refactoring, and design pattern, because we treat refactoring and design patterns as special types of refinement.

In this article we describe an approach to support, in a (semi-) automated way, traceability analysis for vertical impact analysis of UML class diagrams and we formalize the notions of traceability (link) and refinement in that specific context. This is motivated by two objectives: (1) To specify, in an unambiguous manner, possible types of refinements and traceability rules, (2) To facilitate automated traceability analysis based on state-of-the-art, industry-strength modeling technology. This article focuses on the traceability analysis part of the problem and leaves VIA for future work. However, it is important to keep in mind that the overall goal is to support VIA as this may drive some of our decisions.

The rest of the article is structured as follows. Section 2 reports on related work. Our approach is described in details in Section 3. Section 4 presents a case study, and Section 5 shortly discusses our prototype tool. Conclusions are drawn in Section 6.

2. Related work

Several streams of research relate to our problem: refinement analysis (Section 2.1); traceability link analysis (Section 2.2); impact analysis (Section 2.3).

2.1. Classifying and identifying refinements

The UML notation can be extended with specific stereotypes[1] to specify refinements [33,43]. For example, a refined operation that accesses a new attribute [33] is a refinement. An association which is refined into two associations and a new class [43] is another refinement example. Similarly, authors suggest that designers use a specific stereotype (≪`refine`≫) to define refinements, and that refined and refinement elements be linked thanks to UML metamodel element `Abstraction` [40]. Despite the benefits, these solutions may however become a burden to designers who have to perform model modifications and document them with the proposed stereotypes.

In [15] the authors are interested in preserving consistency during the evolution of UML-RT [42] models, a variation of UML specifically dedicated to real-time systems. The authors identify three kinds of modifications (Creation, Deletion, and Update), and focus on four main model elements of UML-RT models (capsules, port, connectors, and protocols). Thanks to a (partial) mapping from UML-RT to Communicating Sequential Processes, the authors are able to identify under which conditions modifications of these model elements maintain consistency, e.g., under which conditions a modified deadlock free model remains deadlock free. The authors restrict their analysis, e.g., to non-hierarchical UML-RT models, and do not show how these atomic changes can be combined into more complex changes and how consistency can then be checked.

In [50] the authors propose an approach to identify refactorings based on the authors' previous work: *UMLDiff* [49]. UMLDiff is used to identify elementary structural changes between two versions of design-level software artifacts such as UML models or code. Four types of structural changes are defined: addition, removal, move, and renaming. Changes to associations are not taken into account. After obtaining a set of elementary structural changes, a set of queries are performed to automatically detect refactorings. These queries are not discussed in the paper and the reader is referred to the implemented software for detailed information about the specification of refactorings and the related detection algorithms. Two taxonomies of refactorings are provided, most of which are from Fowler's work [18].

A classification of nine class diagram refinements is presented in [21], e.g., adding an attribute to a class; splitting a class into

[1] Stereotypes are specified in UML for the purpose of extending the semantics of existing model elements without changing their structures [38]. For example, stereotype ≪`refine`≫ is a standardized stereotype, which "specifies a refinement relationship between model elements at different semantic levels, such as analysis and design".

two classes with an association. These structural refinements are then used to specify behavioral refinements in collaboration diagrams. Only a few simple refinements are presented though and their detection by a tool is not described.

A number of class diagram refinements are discussed in [14]. Their (semi-)automated identification however requires that traceability links be already established. (We will see that our approach, on the other hand, does not make this assumption.) Once traceability links are known, refinements are used to abstract out information from the refined model and reconcile this information with the abstract model. Those refinements are based on the topology (e.g., associations between classes) of the UML class diagrams (abstract, refined) being analyzed, whereas our approach also relies on the automated detection of refinements and their intent, which is modeled by a taxonomy of refinements.

An interesting approach towards the specification of UML refinements is suggested in [39]. The author first formally specifies legal refinements in Object-Z and then studies how these Object-Z refinements translate into UML. No identification mechanism of such refinements is suggested though.

In terms of refinement classification, there is no related work that proposes a systematic and precise classification. Some of the relevant articles [41,42] classify refinement at a very low level (e.g., addition and deletion of a model element); others only take into account a few simple refinements which are not organized in a systematic way [14,21]. Another body of work only focuses on refactoring (a special type of refinement), in particular those defined by Fowler [18]. In terms of refinement identification/detection, some related works rely on stakeholders to explicitly specify refinements by means of stereotypes; some related works do not address the automated identification of refinement at all. Our refinement taxonomy is based on a careful and systematic study of the related literature or according to our own experience of UML-based software development. It contains 31 concrete class diagram atomic refinements and is also expected to be refined over time. Besides, each concrete refinement in the taxonomy is rigorously specified, following a template description.

2.2. Capturing traceability information

In [1], the authors review the current state of the art of model traceability and also mention open issues like automatic traceability link creation. As indicated in the paper, there are three research directions for the automatic traceability link creation and maintenance. One is to apply text mining and information retrieval techniques to recover traceability links between software artifacts (e.g., [2,20]). In this line of work, traceability links are generated through computing a similarity score between a query and each artifact in a set of software artifacts. The limitation of such information retrieval techniques is that they are probabilistic in nature and they will never provide perfectly accurate result (100% recall and precision). As discussed in [32], in practice, typical recall and precision values are around 70% and 30%, respectively, which means that software engineers are required to manually analyze and discard a high number of false positives (due to low precision). A second research direction is to derive traceability links from existing ones. This kind of approaches requires the existence of a set of initial traceability links between software artifacts. In [36], an approach for managing the evolution of traceability links between a software architecture and its implementation is presented. When an architecture or its corresponding source code evolve, existing traceability links are analyzed and updated. The granularity of this approach is coarse-grained and its implementation works at the java file level. Establishing traceability links by monitoring users' modifications and analyzing change history is a third research direction. Our approach fits into this category. We establish traceability links between model elements belonging to two different models by identifying refinements from a set of atomic changes obtained by monitoring user's modifications.

Letelier presents a framework for the specification of traceability links between high-level requirements and UML models [29]. Although primarily intended to support stakeholders in tracing high-level (textual) requirements to various UML models during initial development phases, the framework also allows traceability links between UML model elements. Defining such traceability links is however the responsibility of the stakeholders. Similar tool support to help engineers and maintainers handle traceability links is also suggested in [48].

Another way to establish traceability links automatically is to rely on automated model transformations. Once one knows how a model is transformed, the transformation can be studied and traceability links can be established between model elements of the two model versions. Such transformations are at the core of MDA [25] and some automated transformations have already been described, for instance for design patterns [24]. The use of such transformations to establish traceability has not received much attention, with the noticeable exception of [35,41,47] where the issue is mentioned.

2.3. Performing impact analysis

Most impact analysis techniques rely on a graph representation of software artifacts (nodes) and their dependencies (edges). This graph can (for instance) represent classes, attributes, and operations (nodes) and definitions of attributes/operations in classes as well as operation calls (edges) [31], or simply class dependencies [27]. These dependencies can be retrieved from the code or from design models. As opposed to the simple traversal of the dependency graph to identify direct and indirect impacts [27], the work in [31] suggests a detailed analysis of the changes, organized in a change taxonomy, to precisely study how changes propagate (if they do). Such a change taxonomy, though more extensive and based on changes and impacts on UML models (rather than primarily on code), is also used in [6]. Rules precisely define under which conditions changes propagate, i.e., under which conditions we obtain indirect impacts.

Note that these strategies (including [6]) have all been defined for horizontal impact analysis. Vertical impact analysis is suggested in [3,33], although this terminology is not used by these authors. No precise vertical impact analysis approach is described though. To the best of our knowledge, no such work has been reported to date. One noticeable exception is [48] where maintaining traceability is restricted to traceability between high level textual requirement descriptions and use cases (and use case descriptions). In the same vein, a framework for the synchronization and evolution of models of varying kinds (i.e., varying metamodels) is proposed in [23]: models to be synchronized are viewed as graphs, specified according to the framework; dependencies between model specifications (graphs) are specified according to the framework (traceability descriptions), and changes to models can then be propagated. The authors, however, assume the models are already synchronized, i.e., traceability links are known, and thus focus on the representation of such links and their use to propagate changes in models whose metamodels might be different.

In [45] the authors propose a qualitative approach to represent and quantify traceability links between design elements and the design rationale for the purpose of performing change impact analysis. A Bayesian Belief Network (BBN) is applied to capture probabilistic, causal traceability links. The change impact analysis mechanism is based on the BBN-based reasoning method. The approach also relies on the existence of traceability links between the design rationale and other software artifacts. Besides, the probabil-

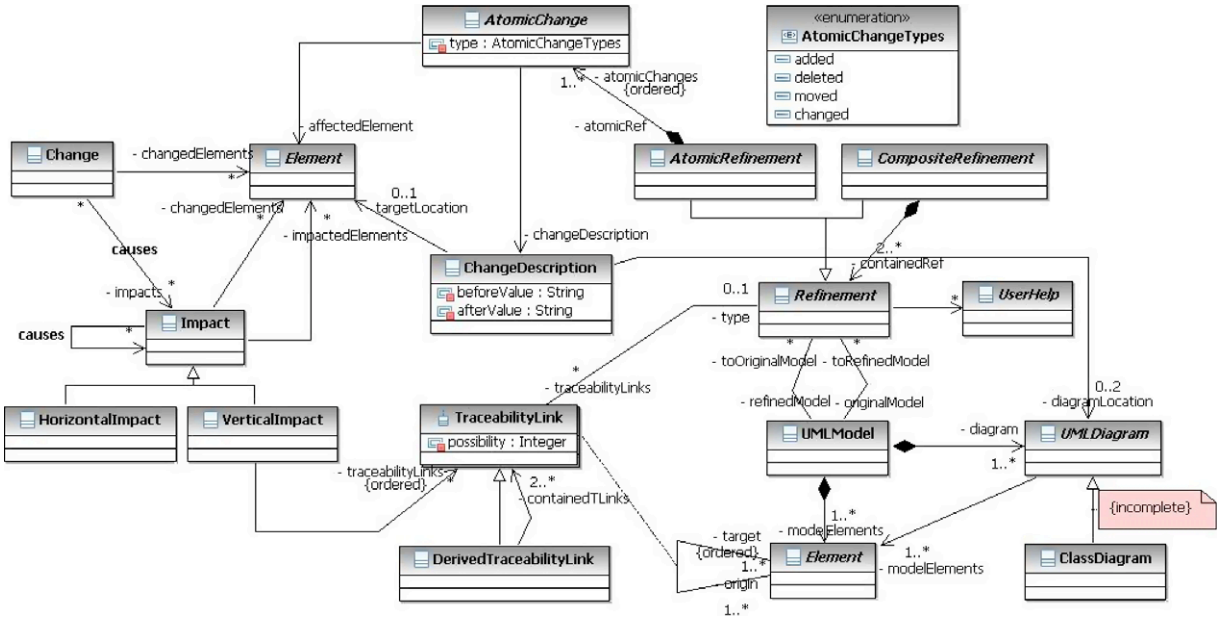

Fig. 2. Conceptual metamodel.

ities on nodes in the BBN model are provided by architects and are estimates based on their experience and intuition. In this article, in order to support vertical impact analysis, we focus on the (semi-) automated identification of traceability links.

3. Traceability links for VIA

Our overall objective is to perform vertical impact analysis (VIA) of UML 2.0 models as automatically and efficiently as possible. As already discussed, this requires that traceability links be established between model elements at the two levels of abstraction during vertical evolution: we need to identify the changes to the analysis model that may lead to changes in the design model. First, atomic changes, which are the elementary steps by which one model evolves into another, are identified automatically. Second, refinements are derived from the identified atomic changes, thus capturing the user's intent at a higher level of abstraction than atomic changes. Third, traceability links are established between model elements of UML model versions automatically (e.g., from analysis model elements to design model elements), based on the identified refinements. Finally, with two UML model versions, the corresponding traceability links, and a HIA approach in hand, VIA can be performed to answer the question: what is the impact of changes to the abstract model on the refined model? We intend to automate the above four activities as much as possible, though the current paper focuses on the first three ones. Note that though our objective is not to require that the designer explicitly specifies refinements, human input might be desirable on occasions. We aim, however, at minimizing the occurrence of such human input.

In the following, we formalize the notions of atomic change, refinement, traceability link, VIA and HIA, by means of a metamodel (Section 3.1). We then present taxonomies of atomic changes and refinements for class diagrams (Sections 3.2 and 3.3). Section 3.4 shows, on one example, how we specify refinements and the corresponding traceability links. A working example is also provided in [7] to illustrate our methodology.

3.1. Metamodel

The class diagram in Fig. 2 illustrates the main concepts of our approach and their relationships. This is also the starting point for the design of our tool presented in Section 5. A data dictionary describing each metaclass, its attribute(s) and association(s) is provided in [7] for reference.

A `UMLModel` is composed of `Elements` (i.e., association `modelElements`), grouped into diagrams (class `UMLDiagram`): this is formally specified by the first Object Constraint Language (OCL) constraint in Fig. 4. Each `UMLDiagram` can be a `ClassDiagram`, or any other valid UML diagram. `Element` is one of the meta-classes of the UML 2.0 metamodel [37], and is therefore the bridge between our metamodel and the UML 2.0 metamodel.[2] A `UMLModel` is associated with instances of `Refinement` by two associations since a model can be refined (rolename `toRefinedModel`) and at the same time be the refinement of another model (rolename `toOriginalModel`). A refinement is between the original model (rolename `originalModel`) and the refined model (rolename `refinedModel`). A refinement is either an `AtomicRefinement`, corresponding to a series of `AtomicChange`s, or a `CompositeRefinement` composed of more than one `Refinement`. `AtomicChange` and `AtomicRefinement` are the roots of hierarchies (taxonomies) of atomic changes and refinements (Sections 3.2 and 3.3). Since the analyzed models may not contain enough information to identify a refinement, the user's help may be requested (class `UserHelp`). If identifying a specific refinement requires user input, we define, in a specific format (e.g., yes-no or multiple-choice questions) a request for information to be presented to the user. Recall that our objective is to minimize the user's involvement.

An `AtomicRefinement` is derived from a group of `AtomicChanges` that occur together (multiplicity 1..* of role name `atomicChanges`). It cannot be decomposed into other refinements. A `CompositeRefinement` consists of more than one `Refinements` which can be `Atomic-Refinements` and/or `CompositeRefinements`. The refinements grouped into a composite refinement are between the same original and refined models. This is specified by the second OCL constraints in Fig. 4.

An `AtomicChange` affects a model element (instance of `Element`) and is further described by a `ChangeDescription`. We

[2] Note that although we focus on refinements and traceability links for class diagrams in this paper, class `Element` will allow us to extend our approach and consider refinements and traceability links for other UML diagrams, as well as ones involving several different UML diagrams at the same time.

consider four different kinds of AtomicChanges, as defined in enumeration AtomicChangeTypes: changed, moved, deleted, and added elements. The last two are self-explanatory: the added (removed) element is the affectedElement in the refined (original) model. An element is moved when its location changes, e.g., an operation is moved from a class to another. For a moved element, affectedElement represents the element in the original model, whereas targetLocation or diagramLocation associations from class ChangeDescription describe the new location. In the case the moved element belongs to another model element (e.g., an operation belonging to a class is moved), targetLocation is to be used. If instead the moved element belongs to a diagram (e.g., a class which belongs to a class diagram is moved to another diagram), diagramLocation is to be used. A changed element occurs when a named element changes name, e.g., an attribute name changes. The name change is then recorded in attributes beforeValue and afterValue of class ChangeDescription. These notions are consistent with the compare and merge facility [30], and the notion of deltas between models which is available in Rational Software Architect (RSA) [22]. The compare and merge facility is indeed the mechanism we intend to rely on for the detection of model changes to identify refinements. Constraints on the metamodel that correspond to these descriptions are given in Fig. 3.

Instances of TraceabilityLink are established for each Refinement instance, between the elements in the original model that are being refined and the elements of the target model that are the refinements (associations to Element with role names origin and target, respectively). The origin and target of a traceability link are model elements of the original and refined models, respectively, of the corresponding refinement. This corresponds to the third constraints in Fig. 4.

When traceability links can be established between a series of models, we can derive traceability links between the elements of the first (most abstract) model and the elements of the last (most refined) model, these links being modeled as instances of class DerivedTraceabilityLink. For instance, if class A in model version 1 is refined into class A in model version 2 which is itself refined into class A in model version 3, the two traceability links (from version 1 to version 2, from version 2 to version 3) can be used to establish a traceability link between A in version 1 and A in version 3, which corresponds to an instance of DerivedTraceabilityLink in our metamodel.

Once refinements between two model versions (say, the original analysis model and the refined design model) have been identified and the corresponding traceability links established, one can perform vertical impact analysis (class VerticalImpact). A VerticalImpact relies on traceability links to identify the elements of the refined model (impactedElements) that may need to be changed because of a set of changes in the original model (changedElements). Since traceability links indicate how each original model element is refined, the changes to the original model are necessarily at the origin of some traceability links. Similarly, the set of elements impacted by the VerticalImpact must be a subset of the target elements of the traceability links. This is modeled by the fourth OCL constraint in Fig. 4.

The changed elements at one level of abstraction (e.g., in the original model) can be the result of a HIA, i.e., a HorizontalImpact can cause another HorizontalImpact. Impacted elements in a more abstract level can be the starting point of a VIA: a HorizontalImpact can cause a VerticalImpact. Similarly, the elements of the refined model being impacted by the VIA can be the starting point of a HIA: a VerticalImpact can cause a HorizontalImpact. In other words, the result of a HIA can be the starting

```
Context ChangeDescription
not(self.beforeValue = self.afterValue)

Context AtomicChange
self.affectedElement->notEmpty()
and  self.type=AtomicChangeTypes::added implies  (
       self.changeDescription->isEmpty()
       and  self.atomicRef.refinedModel.diagram.modelElements
              ->includes(self.affectedElement)
       and  self.atomicRef.originalModel.diagram.modelElements
              ->excludes(self.affectedElement)
     )
and  self.type=AtomicChangeTypes::deleted implies  (
       self.changeDescription->isEmpty()
       and  self.atomicRef.originalModel.diagram.modelElements
              ->includes(self.affectedElement)
       and  self.atomicRef.refinedModel.diagram.modelElements
              ->excludes(self.affectedElement)
     )
and  self.type=AtomicChangeTypes::changed implies  (
       self.changeDescription->notEmpty()
       and  self.changeDescription.targetLocation->isEmpty()
       and  self.changeDescription.diagramLocation->isEmpty()
       and  self.atomicRef.originalModel.diagram.modelElements
              ->includes(self.affectedElement)
       and  self.atomicRef.refinedModel.diagram.modelElements
              ->excludes(self.affectedElement)
     )
and  self.type=AtomicChangeTypes::moved implies  (
       self.changeDescription->notEmpty()
       and  if(self.affectedElement.oclIsTypeOf(Classifier))  (
              self.changeDescription.diagramLocation->notEmpty()
              and  self.changeDescription.targetLocation->isEmpty()
            )  else  (
              self.changeDescription.diagramLocation->isEmpty()
              and  self.changeDescription.targetLocation->notEmpty()
            )
            endif
       and  self.atomicRef.originalModel.diagram.modelElements
              ->includes(self.affectedElement)
   )
```

Fig. 3. Constraints of the metamodel for atomic change.

```
Context UMLModel
self.modelElements->includesAll(self.diagram.modelElements)

Context CompositeRefinement
self.containedRef->forAll(n:Refinement| n.originalModel = self.originalModel
                                    and n.refinedModel = self.refinedModel)

Context TraceabilityLink
self.type.originalModel.diagram.modelElements->includes(self.origin)
and self.type.refinedModel.diagram.modelElements->includes(self.target)
and self.type.traceabilityLinks->includes(self)

Context VerticalImpact
self.traceabilityLinks.type.originalModel->asSet()->size()=1
and self.traceabilityLinks.type.refinedModel->asSet()->size()=1
and self.traceabilityLinks.origin->asSet()->includesAll(self.changedElements)
and self.traceabilityLinks.target->asSet()->includesAll(self.impactedElements)
```

Fig. 4. Some OCL constraints on the metamodel.

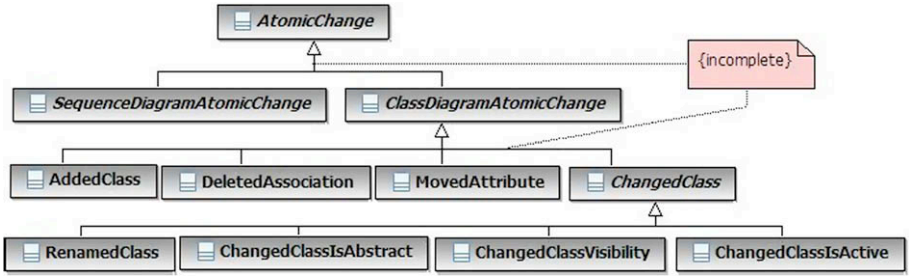

Fig. 5. Taxonomy of atomic changes.

point of a VIA and a VIA can cause another HIA. This is modeled by the reflexive association on class `Impact`.[3] HIA is modeled by classes `Change` and `Impact`, and their associations, which we reuse from an earlier work on UML-based impact analysis [6]. We introduce two classes `VerticalImpact` and `HorizontalImpact` as the subclasses of `Impact` in order to distinguish HIA impacts from VIA impacts.

3.2. Taxonomy of atomic changes

In order to precisely specify (atomic) refinements we need to precisely specify the atomic changes they can be composed of. Such a taxonomy was presented in [6] for HIA purposes, where each UML model element was defined by a set of properties (e.g., a class has attributes) among which core properties uniquely identify the element (e.g., the class name).

Since vertical evolution involves model changes, we adapted the taxonomy presented in [6] for the purpose of identifying traceability links, while accounting for the fact that we rely on the compare& merge facility of RSA to provide such atomic changes. First, in [6], changing the core properties of an element is interpreted as the deletion of the element and the addition of a new element. This is however classified as a change by RSA [22,30], i.e., as an atomic change of type `changed`. Second, moving an element from a location to another (e.g., an operation from a class to another) is classified as a deletion and an addition in [6], whereas in RSA we have the notion of moved elements. In other words, the taxonomy was adapted to benefit from the precise, fine-grained identification of changes provided by RSA.

Though an `AtomicChange` can occur in any UML diagram, we focus in this paper on class diagrams. Fig. 5 shows an excerpt of the taxonomy: an `AtomicChange` in a class diagram can be the addition of a class, the deletion of an association, the move of an attribute between two classes or the change of a class, which is either a change of name, a change of property `IsAbstract` [37] and so on. The complete taxonomy for class diagram atomic changes currently contains 47 concrete changes [7], i.e., leaf nodes in the inheritance hierarchy rooted in `ClassDiagramAtomicChange`, and can be refined over time if necessary.

3.3. Taxonomy of refinements

We also define a taxonomy of atomic refinements (Fig. 6), again focusing on class diagram refinements in this paper. Notice that using the composite design pattern we model the possibility that a refinement involves both class and sequence diagram atomic refinements. (Other diagrams could be added to Fig. 6 using the same principle.) We defined our taxonomy of class diagram atomic refinements based on a careful and systematic study of the related literature (conference and journal articles (e.g., [13]), text books (e.g., [9,10,18,28]), and web pages (e.g., [17])) and according to our own experience of UML-based software development. This taxonomy is also expected to be refined over time.

Fig. 6 shows an excerpt of our taxonomy of 31 concrete class diagram atomic refinements, i.e., leaf nodes in the inheritance hierarchy rooted in `ClassDiagramRefinement` [7]. As an example, `Class -> Classes+Rels` refers to a family of refinements where a class is refined into a set of classes and their relationships. For instance, `TopDownGen` is a refinement through generalization: the class being refined is added a subclass in the refined model. Other refinements, not shown in Fig. 6, describe the refinement of classes and their relationships into a single class or the refinement of a class into several classes (e.g., splitting of responsibilities) [7].

3.4. Refinement/traceability link specification

Each concrete refinement in the taxonomy is rigorously specified, following a template description containing: a general description of the refinement; the list of atomic changes that must be present, and the constraints they must satisfy, to

[3] VIA and HIA are therefore accounted for in our traceability definitions, although the details of the impact rules being used during VIA and HIA are left for future work.

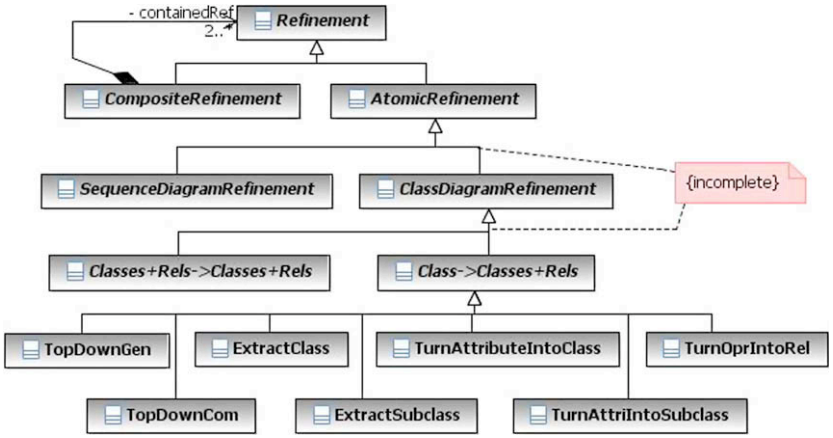

Fig. 6. Taxonomy of refinements.

unambiguously identify the refinement; and a description of the corresponding traceability links. Constraints on atomic changes and traceability links are described using the OCL. We believe that, if we want such research to converge over time and other researchers to build on it, the method employed for detecting changes, refinements, and establishing traceability links needs to be rigorously specified.

As an example, consider the class diagram atomic refinement `TopDownGen` (Fig. 6) where a class is refined by means of a top-down generalization: a subclass is added to this class. The specification of refinement `TopDownGen`, i.e., the characterization of its atomic changes and their relations, is shown as the OCL invariant for class `TopDownGen` in Fig. 7a. The expression indicates that a `TopDownGen` refinement is composed of two atomic changes of type `AddedClass` and `AddedGeneralization` (first three terms in the conjunction). Additionally, the newly added generalization relationship relates the added class (the child class of the generalization) and a class that belongs to the original model (last two conjuncts).

For such a refinement, two traceability links have to be established: one between the original class and the superclass of the added generalization; one between the original class and the added subclass. The rationale is that if a change to the original class (in the original model) is performed, or if this class is deemed to be impacted based on horizontal impact analysis, then both the parent class and the child class of the added generalization (in the target model) may need to be updated. (Recall that how the traceability links are actually used during VIA is outside the scope of this paper.) These traceability links are specified in the OCL expression shown in Fig. 7b. The first four `let` expressions define local variables for the rest of the constraint: `newGen` refers to the added generalization; `superClass` and `subClass` refer to the parent and child classes of the added generalization, respectively; and `origClass` refers to the class in the original model that is being refined, and is identified by looking for the class in the original model which has the same name as the superclass. Using these local variables, the rest of the expression (after the in keyword) indicates that there are two traceability links for the `TopDownGen` refinement, one between `origClass` and `superClass`, and one between `origClass` and `subClass`.

3.5. Algorithm for identifying refinements and establishing traceability links

For the specification of atomic refinements, we have one alternative: to assume that each refinement is specified independently of all the other possible refinements that can apply to the refined

```
Context TopDownGen
let newGen = self.atomicChanges->select(oclIsTypeOf(AddedGeneralization)).
             affectedElement.oclAsType(Generalization)
in
self.atomicChanges->size() = 2
and self.atomicChanges->exists(oclIsTypeOf(AddedClass))
and self.atomicChanges->exists(oclIsTypeOf(AddedGeneralization))
and newGen.specific->includes(self.atomicChanges->
      select(oclIsTypeOf(AddedClass)).affectedElement)
and self.originalModel.diagram->select(oclIsType(ClassDiagram)).modelElements->
        select(oclIsTypeOf(Class)).name->includes(newGen.general.name)
```

(a) Constraint on atomic changes

```
Context TopDownGen
let newGen = self.atomicChanges->select(oclIsTypeOf(AddedGeneralization)).
     affectedElement.oclAsType(Generalization)
let superClass = newGen.general
let subClass = newGen.specific
let origClass = self.originalModel.diagram->
     select(oclIsTypeOf(ClassDiagram)).modelElements->
       select(oclIsTypeOf(Class))->select(name = superClass.name)
in
self.traceabilityLinks->size() = 2
and self.traceabilityLink->exists(origin = origClass and target = superClass)
and self.traceabilityLink->exists(origin = origClass and target = subClass)
```

(b) Traceability links

Fig. 7. Refinement TopDownGen.

```
Algorithm RefIdentification&Traceability(observedACs, originalM, refinedM, refRepository) : traceabilities
Input    observedACs:Sequence(AtomicChange) --- A sequence of observed atomic changes in the order they are observed.
         originalM:UMLModel --- The original UML model
         refinedM:UMLModel --- The refined UML model
         refRepository:Set(Refinement) --- The repository containing all refinement specifications
Output   traceabilities:Set(TraceabilityLink) ---A set of identified traceability links
Declare  model1:UMLModel --- A UML model
         model2:UMLModel --- A UML model
         intermediateMs:Sequence(UMLModel) --- A sequence of intermediate models
         identifiedRef:Refinement --- An identified refinement
         remainingACs:Sequence(AtomicChange) --- A sequence of atomic changes left to be traversed
         done:Boolean --- A Boolean variable
Begin
1    model1 ← originalM
2    intermediateMs ← {}
3    remainingACs ← observedACs
4    done ← false
5    Repeat
6        model2 ← null
7        identifiedRef ← IdentifyRefinement(remainingACs, refRepository, model1, matchingACs)
8        If (identifiedRef != null)
9            remainingACs.remove(matchingACs)
10           model2 ← ApplyRefinement(identifiedRef, model1)
11           EstablishTraceabilities(identifiedRef, model1, model2)
12           intermediateMs.add(model2)
13           model1 ← model2
14       Else
15           done ← true
16       End If
17   Until done
18   traceabilities ← DeriveTraceabilityLinks(originalM, intermediateMs, refinedM)
19   return traceabilities
End
```

Fig. 8. Refinement identification and traceability algorithm.

element(s), or to account for all those other possible refinements during specification. We have opted for the former as this simplifies the specification expressions without any loss of precision. Taking the `TopDownGen` refinement of Fig. 7 as an example, the reader has noticed that the parent class of the added generalization (i.e., in the refined model) is identified in the original model by its name, i.e., we assume the name of the class being refined by `TopDownGen` has not changed. Therefore, for specification purposes only, we assume that the class being refined with the `TopDownGen` refinement is not also refined according to another refinement, e.g., its name does not change as a result of a refinement and the class is not split into two classes as a result of a refinement. Alternatively, accounting for every possible other refinement would require that we account for a possible change of name, a possible splitting of the class and many other refinement types in the OCL expression of Fig. 7. This would clearly make this expression overly complex and would limit the practicality of the approach.

Instead of accounting for multiple refinements (and associated traceability links) in refinement specifications, we rely on an algorithm (Fig. 8) to avoid the situation where multiple refinements occur sequentially and are required to be specified together. The algorithm is formalized using pseudocode where some variables are typed as model elements in our metamodel (Fig. 2), which are highlighted in `courier` font. The algorithm relies on the fact that we collect atomic changes in the order they are performed by the designer, thanks to the compare and merge facility of RSA.[4] The inputs are therefore the original and refined models between which traceability links have to be established (parameters `originalM` and `refinedM`, respectively), a collection of refinement specifications (parameter `refRepository`), and a sequence of observed atomic changes (parameter `observedACs`). The problem bears some similarities with finding patterns, i.e., refinements' sets of atomic changes as defined in `refRepository`, in the input sequence `observedACs`. However, one major difficulty (compared to string pattern matching) is that the atomic changes specifying a given refinement may not necessarily appear in a specific order in the input sequence (the order of appearance of atomic changes in `observedACs` is not relevant to identify an occurrence of the refinement/pattern), and atomic changes specifying different refinements may be mixed/interleaved (the successiveness in `observedACs` of the atomic changes specifying a refinement is not relevant to identify an instance of the refinement).

At the core of the algorithm is function `IdentifyRefinement(seqAC, refs, m, idAC)`: line 7. This function has three input parameters: a sequence of atomic changes to identify refinement from (`seqAC`), a set of refinement specifications (`refs`), and a model (`m`). It returns zero or one refinement identified in `seqAC`, and has one output parameter (`idAC`), namely the set of atomic changes that correspond to the identified refinement. Starting from the first atomic change in `seqAC`, this function tries to identify the smallest sub-sequence of `seqAC` in which a refinement from `refs` can be identified. This relies on the fact that all our refinements have different specifications: they either are derived from different types of atomic changes, different numbers of atomic changes, or the atomic changes must satisfy different (OCL) conditions. It is therefore sufficient to look for the smallest sub-sequence. Note though that if the identified refinement requires n atomic changes, this sub-sequence may have more than n atomic changes (though only n of them will be matched) since refinements may be interleaved.

The algorithm then repeatedly identifies one refinement at a time by calling `IdentifyRefinement()` (loop starting and ending at lines 5 and 17) until no more refinement can be identified (`IdentifyRefinement()` returns null). Each time a refinement is found (lines 9-13), the matched atomic refinements (returned in the last argument of `IdentifyRefinement()`) are removed

[4] The compare and merge facility of RSA can keep track of changes performed by users and the order of these atomic changes.

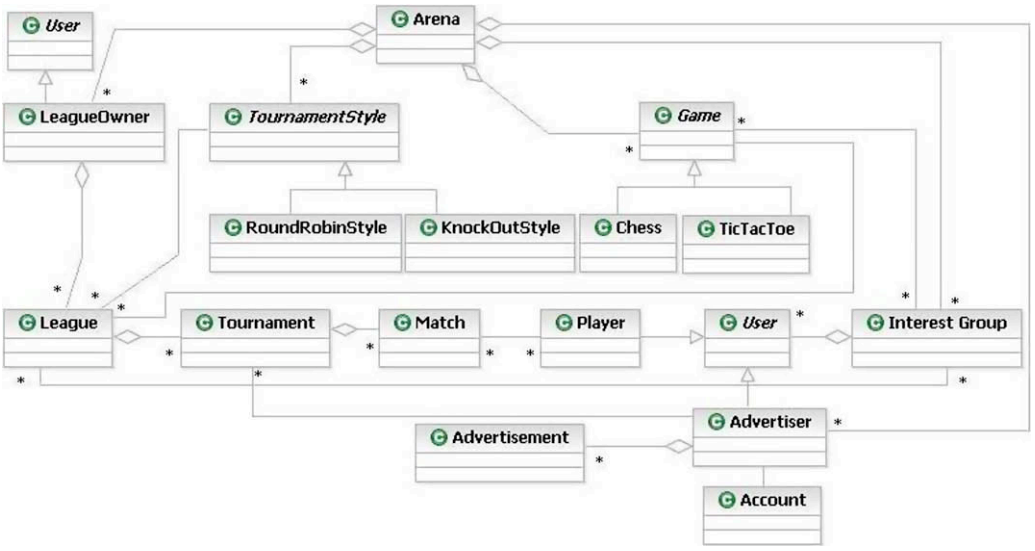

Fig. 9. Case study (from[8]): Analysis model (class `User` appears twice for layout purposes).

from the atomic changes to be used in the next iteration of the loop (line 9). Additionally, an intermediate model is built (line 10) by applying the identified refinement (i.e., by applying the matched atomic changes) and this intermediate model is used during the next iteration of the loop. This way, in subsequent iterations of the loop, we can identify new refinements that are made of atomic changes on previously changed elements, e.g., a class is renamed in a first refinement and then is extended in a second refinement. Traceability links are then established (line 11) between the model before the matched atomic changes have been performed (`model1`) and the model constructed by applying the refinement (`model2`). Once no more refinement can be identified, all the intermediate models and traceability links are revisited (line 18) to establish traceability links between the original model (`originalM`) and the refined model (`refinedM`).

Note that at the end of the execution of the algorithm, some atomic changes may remain unmatched, i.e., the set `remainingACs` may not be empty. This happens if the refined model `refinedM` is an intermediate model for the designer who has additional modifications to be made.

4. Case study

The goal of our case study is to validate the completeness and correctness of our approach and provide illustrative examples of refinement rules.

4.1. Case study subject

Arena is a non-trivial, textbook case study [9]. Its goal is to illustrate object-oriented software development, including requirements elicitation, analysis, system design, and object design. Its analysis and (incomplete) design class diagrams can be found in [9]. We reverse-engineered the complete design class diagram from the code posted on the textbook companion web site [8]. The analysis class diagram is made of 17 classes (Fig. 9), whereas the design class diagram contains 19 classes and 6 interfaces (Fig. 10). Additional characteristics of the two class diagrams are summarized in Table 1. We have omitted all the classes in charge of network communications, data storage, and graphical user interface from the design class diagram since, though part of the design, they are not produced through refinements of analysis classes but are rather added, for example, to address non-functional requirements. These classes therefore do not pertain to our study. Specific refinements and traceability links will need to be devised for such additions, and this is outside the scope of this paper.

A number of reasons led us to select Arena as a case study: (1) it was developed independently from our research, (2) it is a working system, (3) it is sufficiently complex to demonstrate the feasibility of our approach but of a manageable size, and (4) as further discussed below, the changes involved are sufficiently varied to investigate whether our approach is sound and our taxonomies are reasonably complete.

4.2. Validation procedure and summary of results

The goal of our validation was two-fold: (1) To assess whether our atomic change taxonomy was complete with respect to our case study: does it accommodate all changes in our case study; (2) To determine whether our refinement rules could account for all atomic changes and whether they led to correct deductions regarding the refinements made to the analysis model to obtain the design model. In order to achieve this goal, the validation procedure depicted in Fig. 11 was performed on the Arena models to validate our traceability analysis methodology.

The validation procedure of Fig. 11 is composed of five steps. After collecting the analysis and design class diagrams discussed above, we manually identified a sequence of atomic changes that would allow us to transform the former into the latter. To do so, we studied the refinement scenario discussed in the textbook [9] (the textbook provides the rationale for the scenario, following a well-defined development process) and identified the atomic changes that would correspond to this scenario. The identified sequence is therefore a minimum sequence[5] of atomic changes that would allow us to transform the analysis class diagram into the design class diagram (step 1). Admittedly, other possible scenarios—leading to the same design model—could be considered: the order according to which atomic changes are applied could be different and, therefore, the set of refinement rules to be applied could differ as well. Table 2 shows all the refinement types identified from the atomic changes (with their types) of the refinement scenario: The first column lists the refinement types; The second column indicates

[5] It is minimum in the sense that it is the shortest route to go from the analysis diagram to the design diagram, as per the refinement scenario described in the textbook [9] (where only necessary transformations are reported).

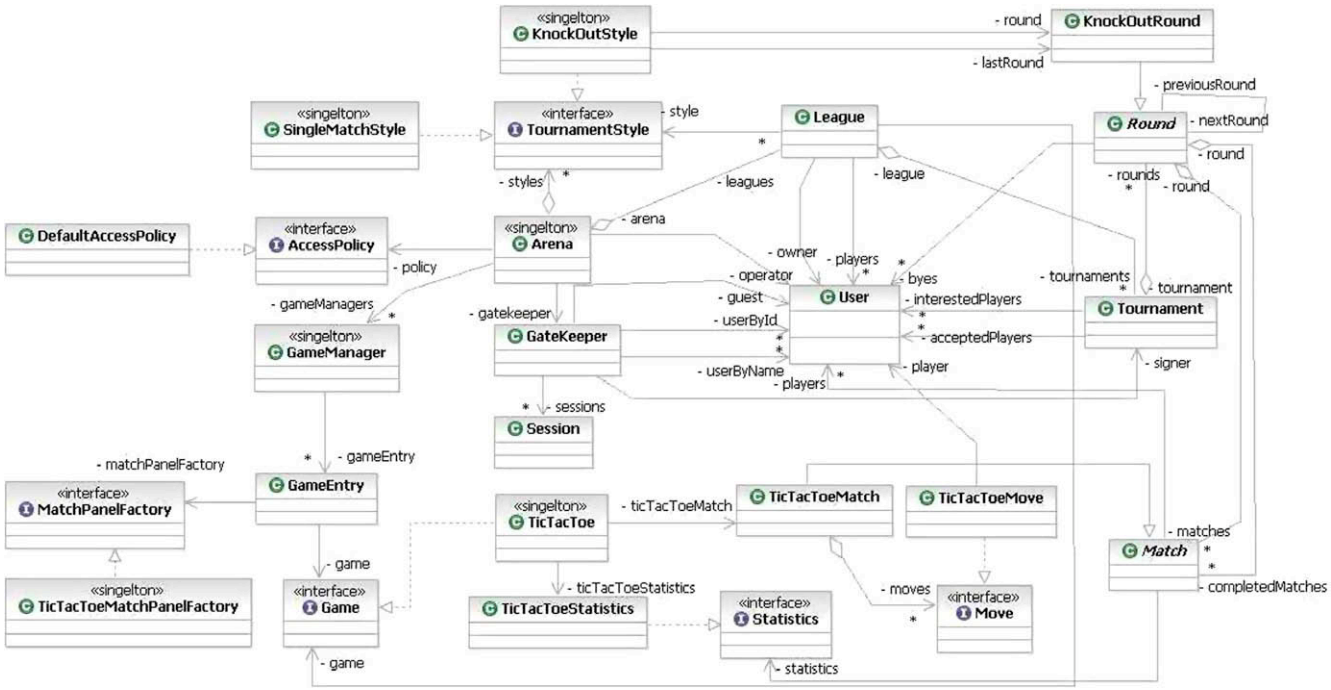

Fig. 10. Case study (from [9]): design model.

Table 1
Class diagrams characteristics

	Class diagram (analysis)	Class diagram (design)
# Classes	17	19
# Interfaces	0	6
# Associations	17	34
# Dependencies	0	7
# Generalizations	7	2
# Model elements	41	68

the number of occurrences of each refinement; Column three shows the actual changes being made to the Arena analysis class diagram; The last column shows the identified atomic changes for the refinements. For instance (first row of Table 2), classes `User` and `Player` of the analysis model have been refined into class `User` of the design model (column 3) after applying a `CollapseHierarchy` refinement (column 1) for which two types of atomic changes are used to identify it (last column): `DeletedClass` and `DeletedGeneralization`; each of them being applied only once. As detailed in Table 2, the selected refinement scenario led to 90 atomic changes, from 12 different atomic change types in our taxonomy.

The second step of our validation was to verify the completeness of our atomic change taxonomy by checking whether the Arena analysis model can be transformed into the design model by only applying the identified atomic changes. If these atomic changes had turned out not to be sufficient to obtain the design model from the analysis model, we would have had to question the completeness of our taxonomy. One important observation from this step was that, for this rather complex scenario, our taxonomy of atomic changes was sufficient to explain how the Arena analysis model can be transformed into the design model.

The third step of our validation was to evaluate our refinement rules (OCL). By analyzing the identified atomic changes, we could devise which refinements were actually performed to obtain the design class diagram. 66 refinements from 18 of the refinement types in our taxonomy [7] were necessary to complete the class diagram transformation from analysis to design. Our refinement taxonomy was therefore deemed complete (step 4) for the case study at hand and it was sufficient to account for all atomic changes identified during step 1. In other words, the 66 identified refinements accommodate all the 90 atomic changes.

The last step was to verify the correctness of the identified refinements by checking their intent, at a high level of abstraction, against the analysis and design class diagrams. Recall that the objective of our approach is to determine the user intent associated with model refinements, which then allows us to establish meaningful traceability links between two models at different levels of abstraction. This step is manual as it requires us to understand why the changes were made and verify that we identified the correct refinements using our rules based on the identified atomic changes. For the Arena case study, our verification is based on the information provided in [9] and our experience regarding object-oriented design. Such analysis showed that the identified refinements were able to explain why and how the Arena analysis model was refined into the design model, while conforming to the information provided in [9]. For example, as shown in the third row of Table 2, we identified an instance of refinement `AddBridgeClass`, which is to refine the association between classes `Tournament` and `Match` in the analysis model into a path between these two classes through a new class `Round`. As indicated in [9] (page 381), the new class `Round` "corresponds to a set of `Matches` that can be held concurrently." The association between classes `Tournament` and `Match` in the analysis cannot model the requirement of organizing a set of matches held concurrently. A new bridge class `Round` has to be added to bridge classes `Tournament` and `Match`. This conforms to the intent of our identified refinement `AddBridgeClass`.

Table 2 therefore shows that all the identified atomic changes are involved in at least one refinement, suggesting that our refinement rules accounted for all atomic changes in the case study: we can fully explain how the analysis model is refined into the design model thanks to the identified atomic changes and the proposed refinement rules. Furthermore, all applied refinement rules were checked for correctness, i.e., the intent of the refinements, at a high level of abstraction, as formalized by the rules, was checked against

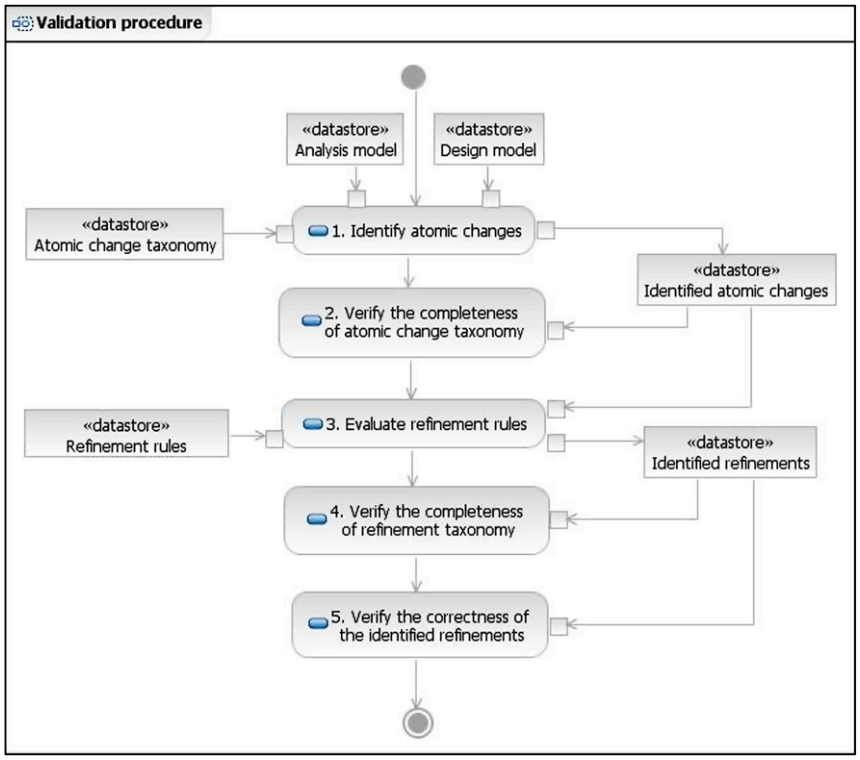

Fig. 11. Validation procedure.

the class diagrams. The goal was to see whether the user's rationale when refining the analysis class diagram into the design class diagram (as explained in the textbook reporting the case study) is captured by the identified refinements.

In summary, the case study shows that all the changes performed to the model belong to our taxonomy of atomic changes, and all those changes are correctly used in at least one refinement rule. In other words, our taxonomies of atomic changes and the set of refinement rules we proposed are complete and correct based on the Arena case study.

4.3. Detailed results

Using excerpts of the analysis and design class diagrams of the Arena case study (Fig. 12), we illustrate in the subsections below two of the refinements (from a total of 66) we have identified, thereby providing additional insights into the results presented in the previous section. Fig. 12a and Fig. 12c are excerpts of the analysis and design class diagrams, respectively. Fig. 12b shows an intermediate step in the refinement scenario discussed previously (recall that it involves several refinements) that we add for illustration purposes.

4.3.1. Refinement detailassofunctionality

Fig. 13 shows the OCL specification of refinement `DetailAssoFunctionality`, whose intent is to refine an association by adding new classes and associations. The general idea for identifying such a refinement is to be able to find, in the refined class diagram, at least one path between the classes at the two ends of the association[6] being deleted in the original model. Referring to Fig. 12a and b, the association between classes `Arena` and `Game` is deleted. These two classes can be found in the refined model, Fig. 12b, and there exists a path between these two classes: class `Arena`, association between `Arena` and `GameManager`, class `GameManager`, association between `GameManager` and `GameEntry`, class `GameEntry`, association between `GameEntry` and `Game`, and class `Game`.

As specified in Fig. 13a, such a refinement is at least made of one `DeletedAssociation` atomic change: the deleted association (in the original model) is referred to as `delAsso`. The next `let` expressions are used to identify the classes at the two ends of the deleted association in the original model (`origA` and `origB`) and the same classes in the refined model (`refA` and `refB`). Last, we use operation `exists()` of class `Path` to identify whether there is a path between `refA` and `refB` in the refined model. Class `Path` does not appear in our metamodel as it is only a helper class used to shorten our OCL expressions. Its specification is given in [7]. A `Path` instance is associated with a sequence of `Elements`, i.e., a path is a sequence of UML model elements. More precisely, elements are classes (or interfaces) and relationships. Operation `exists(endA,endB)`, `endA` and `endB` being classes of the same model, returns true if and only if there exists a path between these two classes. Recall that we assume that only one refinement is applied to the original model when specifying refinements. We can therefore identify a refined class (referred to as `refA` or `refB`) by looking for the class in the refined model which has the same name as the original class (referred to as `origA` or `origB`).

Fig. 13b specifies how traceability links are established for refinement `DetailAssoFunctionality`. For each path that we can find in the refined model between `refA` and `refB` (we use operation `getPaths()` of class `Path`), we establish a traceability link between the removed association `delAsso` (in the original model) and each element of the identified path.

4.3.2. Refinement replacesuperclasswithinterface

The `ReplaceSuperclassWithInterface` refinement consists in replacing a parent class with an interface and replacing

[6] In this article, we assume associations are binary, although n-ary associations can be transformed into binary associations and handled following the same principles. Future work will adapt our OCL expressions to account for n-ary associations.

Table 2
Summary of refinements and atomic changes

Refinement type (total 18)	No. of occurrences	Changes made to the analysis mode[a]	Atomic Changes (for one instance of a refinement type) with no. of occurrences
(Part A)-Summary of refinements and atomic changes			
`CollapseHierarchy`	2	Classes `User` and `Player` -> class `User` Classes `User` and `LeagueOwner` -> class `User`	`DeletedClass` (1) `DeletedGeneralization` (1)
`ReplaceSuperclassWithInterface`	2	Superclass `Game` -> interface `Game` Superclass `TournamentStyle` -> interface `TournamentStyle`	`DeletedClass` (1) `AddedInterface` (1) `DeletedGeneralization` (1) `AddedDependency` (1)
`AddedBridgeClass`	1	Association between `Tournament` and `Match` -> new class `Round` and associations	`DeletedAssociation` (1) `AddedAssociation` (3) `AddedClass` (1)
`TopDownGen`	2	Class Match -> classes `Match` and `TicTacMatch` Class `Round` -> classes `Round` and `KnockOutRound`	`AddedClass` (1) `AddedGeneralization` (1)
`InlineClass`	1	Classes `Interest Group` and `User` -> class `User`	`DeletedClass` (1) `DeletedAssociation` (1)
`DetailAssoFunctionality`	1	Association between `Arena` and `Game` -> path between `Arena` and `Game`	`DeletedAssociation` (1) `AddedAssociation` (3) `AddedClass` (2)
`ByBridgeClass`	1 1	Association between `Arena` and `Interest Group` -> path between `Arena and User` via bridge class `GateKeeper` Association between `Arena` and `LeagueOwner` -> path between `Arena` and `User` via bridge class `League` (rolename `owner`)	`AddedAssociation` (3) `DeletedAssociation` (1)
`ClassIsAbstractRef`	2	Class `User`: abstract -> concrete Class `Match`: concrete -> abstract	`ChangedClassIsAbstract` (1)
(Part B)-Summary of refinements and atomic changes			
`AssoEndMultiplicityRef`	2	Multiplicity between `League` and `TournamentStyle`: many -> 1	`ChangedAssoEndMultiplicity` (1)
`AssoEndIsNavigableRef`	3	Association between `League` and `TournamentStyle`: non-navigable -> navigable	`ChangedAssoEndIsNavigable` (1)
`RelocateAssociationRef`	3	Association between `Match` and `Player` -> association between `Match` and `User`	`RelocateAssociation` (1)
`AddedClassRef`	7	Class `Session` is added	`AddedClass (1)`
`AddedInterfaceRef`	4	Interface `Statistics` is added	`AddedInterface` (1)
`AddedAssociationRef`	16	Association between `Round` and `User` is added	`AddedAssociation` (1)
`AddedDependencyRef`	5	Dependency between `Move` and `TicTacToeMove` is added	`AddedDependency` (1)
`DeletedClassRef`	5	Class `Advertiser` is deleted	`DeletedClass` (1)
`DeletedGeneralizationRef`	3	Generalization between `Game` and `Chess` is deleted	`DeletedGeneralization` (1)
`DeletedAssociationRef`	5	Association between `Advertiser` and `Account` is deleted	`DeletedAssociation` (1)

[a] This column only shows sample examples of refinements from the Arena case study. The complete list of changes and refinements made to the analysis model is provided in [7].

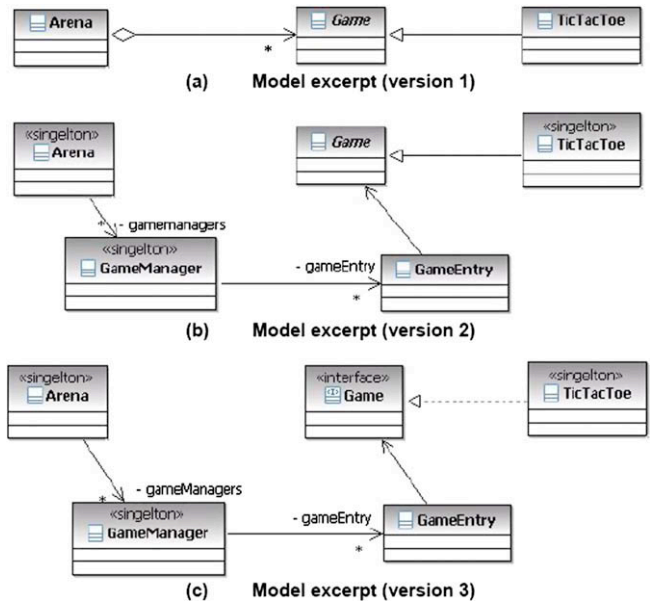

Fig. 12. Two example refinements from the Arena case study.

the generalization relationships (to the replaced class) with implementation relationships. Four kinds of atomic changes are therefore used to identify this refinement: `DeletedClass`, `AddedInterface` (with the same name as the deleted class), `DeletedGeneralization` (its general end is the deleted class), and `AddedDependency` (between subclasses of the deleted class and the added interface). Recalling that in UML 2.0, an implementation is a dependency, these conditions are specified in the OCL expression of Fig. 14a. Notice that, once again, we use class names to recognize classes that belong to both the original and refined models.

When a `ReplaceSuperclassWithInterface` refinement is identified, a number of traceability links are established as specified in Fig. 14b. First, we establish a link between the deleted class (in the original model) and the added interface (in the refined model). Second, we establish a link between the deleted class and each class that implements the added interface. The rationale is that a change to the (deleted) class in the original model may impact the (added) interface as well as all the classes that implement the interface (in the refined model).

Referring to our Arena case study, abstract class `Game` (Fig. 12b) is transformed into an interface (Fig. 12c). The generalization between class `TicTacToe` and class `Game` is transformed into an implementation between class `TicTacToe` and interface `Game`. In this example, two traceability links are established: one from class `Game` (original model) to interface `Game` (refined model); one from class `Game` (original model) to class `TicTacToe` (refined model).

```
Context DetailAssoFunctionality
let delAsso = self.atomicChanges->select(oclIsTypeOf(DeletedAssociation))
    .affectedElement.oclAsType(Association)
let origA = delAsso.memberEnd.class->any(true)
let origB = delAsso.memberEnd.class->excluding(origA)
let refA = self.refinedModel.diagram->select(oclIsTypeOf(ClassDiagram)).modelElements->
    select(oclIsType(Class))->select(name=origA.name)
let refB = self.refinedModel.diagram->select(oclIsTypeOf(ClassDiagram)).modelElements->
    select(oclIsType(Class))->select(name=origB.name)
in
self.atomicChanges->exists(oclIsType(DeletedAssociation))
and Path.exists(refA, refB)
```

(a) Constraint on atomic changes

```
Context DetailAssoFunctionality
let delAsso = self.atomicChanges->select(oclIsTypeOf(DeletedAssociation)).
    affectedElement.oclAsType(Association)
let origA = delAsso.memberEnd.class->any(true)
let origB = delAsso.memberEnd.class->excluding(origA)
let refA = self.refinedModel.diagram->select(oclIsTypeOf(ClassDiagram)).modelElements->
    select(oclIsType(Class))->select(name=origA.name)
let refB = self.refinedModel.diagram->select(oclIsTypeOf(ClassDiagram)).modelElements->
    select(oclIsType(Class))->select(name=origB.name)
in
Path.getPaths(refA,refB)->forAll(p:Path|Sequence{1..p.elements->size()}->
  forAll(i|self.TraceabilityLink->exists(t|t.origin = delAsso
      and t.target = p.elements->at(i))))
```

(b) Traceability links

Fig. 13. Refinement `DetailAssoFunctionality`.

```
Context ReplaceSuperclassWithInterface
self.atomicChanges->size() = self.atomicChanges
  ->select(oclIsTypeOf(AddedDependency))->size()*2+2
and  self.atomicChanges->select(oclIsTypeOf(DeletedClass))->size() = 1
and  self.atomicChanges->select(oclIsTypeOf(AddedInterface))->size() = 1
and  self.atomicChanges->exists(oclIsTypeOf(DeletedGeneralization))
and  self.atomicChanges->exists(oclIsTypeOf(AddedDependency))
and  self.atomicChanges->select(oclIsTypeOf(AddedDependency))->size()
  = self.atomicChanges->select(oclIsTypeOf(DeletedGeneralization))->size()
and  self.atomicChanges->select(oclIsTypeOf(AddedDependency))->
  forAll(oclIsTypeOf(InterfaceRealization))
and  self.atomicChanges->select(oclIsTypeOf(AddedDependency)).supplier
  ->asSet() = self.atomicChanges->select(oclIsTypeOf(AddedInterface))->asSet()
and  self.atomicChanges->select(oclIsTypeOf(DeletedGeneralization)).general->asSet() =
    self.atomicChanges->select(oclIsTypeOf(DeletedClass))->asSet()
and  self.atomicChanges->select(oclIsTypeOf(AddedDependency))->asSet().client.name =
  self.atomicChanges->select(oclIsTypeOf(DeletedGeneralization))->asSet().specific.name
```

(a) Constraint on atomic changes

```
Context ReplaceSuperclassWithInterface
let delClass = self.atomicChanges->select(oclIsTypeOf(DeletedClass))
  .affectedElement.oclAsType(Class)
let newInterface = self.atomicChanges->select(oclIsTypeOf(AddedInterface))
  .affectedElement.oclAsType(Interface)
in
self.traceabilityLinks->size() = self.atomicChanges->
  select(oclIsTypeOf(DeletedGeneralization))->size()+1
and  self.traceabilityLinks->exists(origin = delClass and target = newInterface)
and  self.atomicChanges->select(oclIsTypeOf(AddedDependency)).client->forAll(c |
  self.traceabilityLinks->exists(origin = delClass and target = c))
```

(b) Traceability links

Fig. 14. Refinement `ReplaceSuperclassWithInterface`.

5. Automation

Our approach is implemented in a tool named `VIATool` (Vertical Impact Analysis Tool[7]) which we built as a set of java plug-ins to the Eclipse platform using IBM Rational Software Architect (RSA) [22]. This tool is architected by integrating the following technologies: Eclipse development platform, EMF, OCL Engine, CompareEMerge Engine, and Eclipse UML. Our goal was to obtain a tool that could be easily extended, would easily accommodate certain types of changes, and that could be adapted to other modeling environments than RSA while taking advantage of the most advanced, available technologies.

As shown in Fig. 15, the tool is composed of two subsystems to first identify refinements and establish traceability links between

[7] Only the traceability link identification is implemented as the VIA part is still ongoing work.

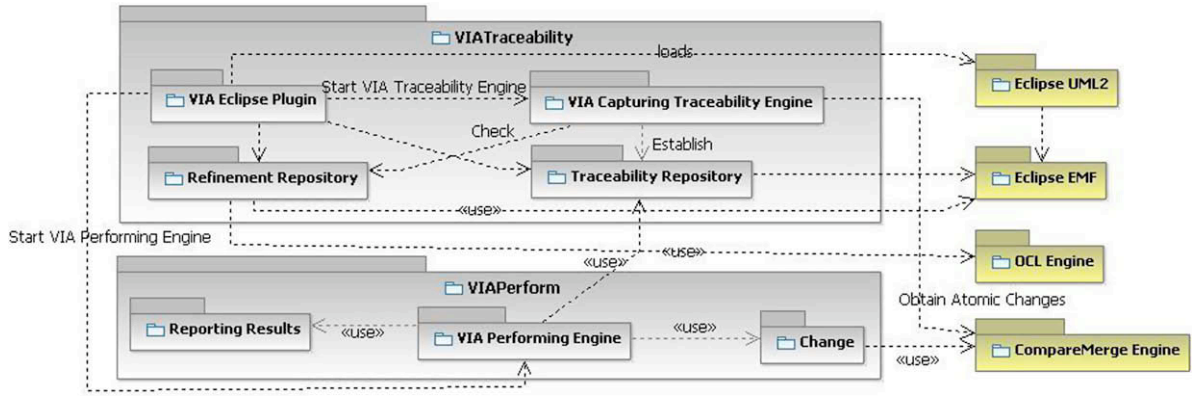

Fig. 15. VIATool architecture overview.

two UML models (subsystem `VIATraceability`), and then perform vertical impact analysis (subsystem `VIAPerform`), though the latter is not described in detail in this paper. Both subsystems rely on existing Eclipse-based technologies: `Eclipse EMF`, `Eclipse UML2`, RSA's `CompareMerge Engine`, and RSA's `OCL Engine`.`Eclipse EMF` [11] is a modeling and code generation engine for building applications based on a structured data model. Using EMF, a developer can define a data model and get a set of Java classes, automatically generated by the EMF facility, to create and manipulate instances of the data model. EMF also produces a set of adapter classes that enable viewing and editing the model, and a basic editor, which is itself an Eclipse plug-in [11]. The `Eclipse UML2` project is an EMF-based implementation of the UML 2.0 metamodel [37] for the Eclipse platform [12], and is integrated into RSA. The UML models to be analyzed are instantiated using the `Eclipse UML2` project and can be obtained from RSA. The `CompareMerge Engine` [30], which is part of RSA, is used to compare two versions of a model. The `OCL Engine` is used to evaluate OCL expressions against an instance of an EMF model.

In the `VIATraceability` subsystem, the `Eclipse EMF` framework is used to create the metamodel of Fig. 2, generate corresponding java APIs, and generate basic user interface (UI) components to create and edit instances of the metamodel. Fig. 16 presents an example of such a UI for editing refinement `TopDownGen`. `Refinement Repository` (Fig. 15) is composed of all the refinements we have specified. `Traceability Repository` (Fig. 15) is used to preserve all the traceability links being established. This repository is then available to `VIAPerform`.

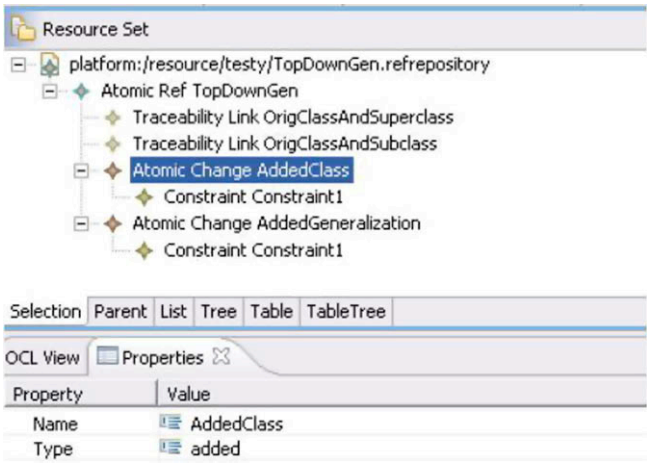

Fig. 16. User interface for editing refinement `TopDownGen`.

In the `VIAPerform` subsystem, the `VIA Performing Engine` is initiated by the `VIA Eclipse Plugin` of the `VIATraceability` subsystem. Vertical impact analysis is triggered by a change (the `Change` subsystem), which is obtained from the RSA's `CompareMerge Engine`. `VIA Performing Engine` performs vertical impact analysis with two UML models, a change, and the established traceability links for these two UML models (`Traceability Repository`) as inputs. Then a report is generated to show the analysis results. Some of the technical details of these subsystems are still under investigation since, as already mentioned, our future work on VIA and HIA will based on the traceability analysis presented in this paper.

Our approach starts from the `VIA Capturing Traceability Engine` that automatically obtains atomic changes. These are the elementary steps by which one model evolves into another, from the RSA's `CompareMerge Engine`. These atomic changes are then used to determine refinement(s) by trying to identify instances of all the refinements available in the `Refinement Repository`, using the corresponding OCL rules (e.g., part (a) of Figs. 7, 13, 14). The `OCL Engine` is used to check those OCL expressions. Then, traceability links are established automatically, based on the identified refinements (e.g., part (b) of Figs. 7, 13, 14). These traceability links are stored in the `Traceability Repository` for later consumption by the `VIA Performing Engine`. When traceability links are ready for two UML models, the `VIA Eclipse Plugin` initiates the `VIA Performing Engine`.

Each step of our methodology can be fully automated. First, our tool relies on RSA's `CompareMerge Engine` to automatically obtain atomic changes. The `CompareMerge Engine` is a mature technology and has been delivered as a component of RSA. Second, the `OCL Engine` is used to automatically query the OCL rules of the refinements in the refinement repository to identify refinements and establish traceability links. The `OCL Engine` is a query framework freely downloadable from the Eclipse's website. Third, a vertical impact analysis can be performed automatically using the traceability links, given a set of changes. The changes triggering the impact analysis are obtained from the `CompareMerge Engine`.

Thanks to the `VIA Capturing Traceability Engine`, our architecture distinguishes the refinement specification (contained in the `Refinement Repository`), specifying required atomic changes, from the observed atomic changes obtained from the `CompareMerge Engine`. As a result, it ensures that, even though our tool relies on a set of existing technologies, they can be replaced with others as long as they provide equivalent functionality. For example, other OCL engines can be used instead of the one provided by Eclipse. Also, the `CompareMerge Engine` can be replaced with other model comparison tools.

Furthermore, all the technologies we used and developed are harmoniously integrated with a user's modeling environment. All of them and the modeling environment are based on EMF and it is therefore easy to integrate them together. For the current implementation, we use IBM's RSA as modeling environment, but our approach is not necessary bound to any special software modeling environment. The only issue we need to consider, when a different modeling environment is used, is that the comparison engine—which operates directly on user models—should be able to provide the atomic changes made to the models developed in the new modeling environment.

As we discussed in Section 3.3, even though our goal was to define a complete refinement taxonomy, it is expected that new types of refinements will be uncovered on new case studies and the taxonomy will consequently need to be refined over time. Our tool is designed to accommodate such changes as users can add their own refinements to the refinement repository by simply creating an instance of the metamodel (corresponding to Fig. 2). The EMF framework automatically generates a basic user interface for the users to create and edit instances of refinements (see Fig. 16).

As we discussed in Section 3, our methodology can be applied to any UML diagram, and not only class diagrams, Though we only provide the taxonomy for class diagram refinements in this paper, our approach and tool architecture are designed to ensure that other UML diagrams can be introduced in the future without any major modifications. This is achieved by storing all the refinements separately in the refinement repository and by ensuring that other components of the tool do not depend on the types of diagrams. For example, the tasks of obtaining atomic changes from the comparison engine or establishing traceability links are all diagram-type independent.

We chose to store and maintain the established traceability links independently from the user's model. There are several benefits in doing so. First we avoid cluttering the user's model and expose all the traceability links, thus adding complexity to the user's modeling activities. Second this solution supports the evolution of the traceability metamodel without requiring changes to other parts of the framework. This may happen when additional information must be captured along with the traceability links, for example probabilities of impact. Since traceability information is independent from the user's model, VIA is performed based on our traceability repository rather than on the user's model. In some cases, if a VIA strategy requires to access the model elements of the user's model, it can simply follow the traceability links which contains references to the user's model they relate.

6. Conclusion

UML models are, when used on typical commercial systems, very complex artifacts. Model-driven development practices [25] are iterative and rely on the stepwise refinement of analysis models into increasingly detailed design models, all the way down to implementation. However, changes (e.g., requirements, architecture) are typically taking place at the same time as model refinements, thus leading to a complex change management problem. How, under those conditions, can one ensure that models at different levels of abstraction remain consistent, especially in a context where changes and refinements are performed by different individuals in the context of large teams?

This paper proposes a methodology and automation strategy to address the above problem. It is based on a careful classification of changes and refinements (capturing the changes' intent at a higher level of abstraction than model element changes) in UML models and an automated identification of refinements based on detected changes. For each refinement, traceability links are then automatically established and can then be used to control the impact of changes in more abstract models on more refined models. We refer to this process as Vertical Impact Analysis (VIA). This work complements and extends earlier work [6] on the impact analysis across UML diagrams within the same model, referred to as Horizontal Impact Analysis (HIA). Another contribution of this paper is of methodological nature: we provide a way, which is based on metamodeling and constraints expressed in the Object Constraint Language, to formally define change types, relate them to change intents (refinements), and formally specify the derivation of traceability links. This approach, while allowing for a rigorous and formal specification, has the advantage of facilitating tool automation (the metamodel and constraints are a starting point for tool design), and is easier to implement using available, industry-strength modeling technology such as Eclipse EMF [11].

A case study, based on a system developed independently from this research, shows that for a non-trivial model refinement scenario, our change taxonomies and refinement rules are complete and correct. They can accommodate the transformation of an analysis model into a design model.

Future work includes further investigating VIA: change impact mechanisms based on traceability links. We foresee that these mechanisms will likely adapt and extend mechanisms already reported for HIA (e.g., [6]). This paper is only a first step as it only addresses traceability analysis to support VIA for class diagrams. More work is under way to extend the work to the complete set of UML diagrams. Furthermore, we need to identify specific traceability mechanisms for design pattern classes and perform additional cases studies.

Acknowledgement

This work is supported by CITO and IBM-Rational. Lionel and Yvan were further supported by NSERC grants.

References

[1] N. Aizenbud-Reshef, B.T. Nolan, J. Rubin, Y. Shaham-Gafni, Model traceability, IBM Systems Journal 45 (3) (2006) 515–526.
[2] G. Antoniol, G. Canfora, G. Casazza, A. De Lucia, E. Merlo, Recovering traceability links between code and documentation, IEEE Transactions on Software Engineering 28 (10) (2002) 970–983.
[3] S.A. Bohner, Software change impacts – an evolving perspective, in: Proceedings of the IEEE International Conference on Software Maintenance, pp. 263–272, 2002.
[4] S.A. Bohner, R.S. Arnold, An introduction to software change impact analysis, in: S.A. Bohner, R.S. Arnold (Eds.), Software Change Impact Analysis, IEEE Computer Society Press, 1996, pp. 1–25.
[5] S.A. Bohner, R.S. Arnold, Software Change Impact Analysis, IEEE Computer Society Press, 1996.
[6] L.C. Briand, Y. Labiche, L. O'Sullivan, M. Sowka, Automated impact analysis of UML models, Journal of Systems and Software 79 (3) (2006) 339–352.
[7] L.C. Briand, Y. Labiche, T. Yue, Automated traceability analysis for UML model refinements, Carleton University, Technical Report SCE-06-06, April, 2006.
[8] B. Bruegge, ARENA, <http://sysiphus.informatik.tu-muenchen.de/arena/index.html> (accessed July 2006).
[9] B. Bruegge, A.H. Dutoit, Object-Oriented Software Engineering Using UML, Patterns, and Java, second ed., Prentice Hall, 2004.
[10] D.F. D'Souza, A.C. Wills, Objects, components, and frameworks with UML: the catalysis approach, Addison-Wesley Longman Publishing Co. Inc., Boston, MA, USA, 1998.
[11] Eclipse Foundation, Eclipse modeling framework, <www.eclipse.org/emf/> (accessed May 2005).
[12] Eclipse Foundation, UML2: EMF-Based UML 2.0 Metamodel Implementation, www.eclipse.org/uml2/ (accessed May 2005).
[13] A. Egyed, Automated abstraction of class diagrams, ACM Transactions on Software Engineering and Methodology (TOSEM) 11 (4) (2002) 449–491.
[14] A. Egyed, Consistent Adaptation and Evolution of Class Diagrams during Refinement, Lecture Notes in Computer Science, vol. 2984/2004, Springer, Berlin/ Heidelberg, 2004.
[15] G. Engels, R. Heckel, J.M. Kuster, L. Groenewegen, Consistency-preserving model evolution through transformations, in: Proceedings of the International Conference on the Unified Modeling Language, LNCS 2460, pp. 212–226, 2002.

[16] G. Engels, J.M. Kuster, L. Groenewegen, Consistent Interaction of software components, in: Proceedings of the Integrated Design and Process Technology, 2002.
[17] M. Fowler, Refactoring Home page, <http://www.refactoring.com/? (accessed May 2008).
[18] M. Fowler, Refactoring – Improving the Design of Existing Code, Addison Wesley, 1999.
[19] O.L. Goaer, P. Ebraert, Evolution styles: change patterns for software evolution, in: Evol 2007. Paris, 2007.
[20] J.H. Hayes, A. Dekhtyar, J. Osborne, Improving requirements tracing via information retrieval, in: Requirements Engineering Conference, 2003, Proceedings. 11th IEEE International, pp. 138–147, 2003.
[21] B. Hnatkowska, Z. Huzar, L. Kuzniarz and L. Tuzinkiewicz, Refinement relationship between collaborations, in: Proceedings of the Workshop on Consistency Problems in UML-based Software Development, UML'03, 2003.
[22] IBM-Rational: Rational Software Architect, 2005. <www-306-ibm.com/software/awdtools/architect/swarchitect/>.
[23] I. Ivkovic, K. Kontogiannis, Tracing evolution changes of software artifacts through model synchronization, in: Proceedings of the IEEE International Conference on Software Maintenance, pp. 252–261, 2004.
[24] S.R. Judson, R.B. France, R.H. Carver, Supporting rigorous evolution of UML models, in: Proceedings of the IEEE International Conference on Engineering of Complex Computer Systems, pp. 128–137, 2004.
[25] A. Kleppe, J. Warmer, W. Bast, MDA Explained – The Model Driven Architecture: Practice and Promise, Addison-Wesley, 2003.
[26] P. Kroll, P. Kruchten, The Rational Unified Process Made Easy: A Practitioner's Guide to the RUP, Addison-Wesley, 2003.
[27] D. Kung, J. Gao, P. Hsia, F. Wen, Y. Toyoshima, C. Chen, Change impact identification in object oriented software maintenance, in: Proceedings of the IEEE International Conference on Software Maintenance, pp. 202–211, 1994.
[28] C. Larman, Applying UML and Patterns, 3rd ed., Prentice-Hall, 2004.
[29] P. Letelier, A framework for requirements traceability in UML-based projects, in: Proceedings of the International Workshop on Traceability in Emerging Forms of Software Engineering, in Conjunction with ASE, 2002.
[30] K. Letkeman, Comparing and merging UML models in IBM rational software architect: Part 3, IBM-Rational, White paper, 2005. <http://www-128.ibm.com/developerworks/rational/library/05/802_comp3/index.html>.
[31] L. Li, A.J. Offutt, Algorithmic analysis of the impact of changes to object-oriented software, in: Proceedings of the IEEE International Conference on Software Maintenance, pp. 171–184, 1996.
[32] A.D. Lucia, F. Fasano, R. Oliveto, Recovering traceability links in software artifact management systems using information retrieval methods, ACM Transactions on Software Engineering and Methodology (TOSEM) 16 (4) (2007) 13.
[33] T. Mens, T. D'Hondt, Automating support for software evolution in UML, Automated Software Engineering (Springer) 7 (1) (2000) 39–59.
[34] T. Mens, T. Tourwe, A survey of software refactoring, IEEE Transaction on Software Engineering 30 (2) (2004) 126–139.
[35] T. Mens, P. Van Gorp, A taxonomy of model transformation, in: Proceedings of the International Workshop Graph and Model Transformation, 2005.
[36] L.G.P. Murta, A. van Der Hoek, C.M.L. Werner, ArchTtrace: policy-based support for managing evolving architecture-to-implementation traceability links, in: Proceedings of the IEEE/ACM International Conference on Automated Software Engineering, pp. 135–144, 2006.
[37] OMG, UML 2.0 Superstructure Specification, Object Management Group, Final Adopted Specification ptc/03-08-02, 2003.
[38] OMG, UML 2.0 Superstructure Specification, 2005.
[39] C. Pons, On the definition of UML refinement patterns, in: Proceedings of the ACM/IEEE International Conference on Model Driven Engineering Languages and Systems (MoDELS) Workshop MoDeVa, 2005.
[40] C. Pons, R.-D. Kutsche, Traceability across refinement steps in UML modeling, in: Proceedings of the Workshop on Software Model Engineering, in conjunction with UML'04, 2004.
[41] I. Porres, Rule-based update transformations and their application to model refactorings, Software and Systems Modeling 4 (2) (2005) 368–385.
[42] B. Selic, Using UML for modeling complex real-time systems, in: Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers and Tools for Embedded Systems, LNCS 1474, pp. 250–260, 1998.
[43] W. Shen, Y. Lu, W.L. Low, Extending the UML Metamodel to support software refinement, in: Proceedings of the Workshop on Consistency Problems in UML-Based Software Development, in conjunction with UML, 2002.
[44] R.V.D. Straeten, V. Jonckers, T. Mens, A formal approach to model refactoring and model refinement, Software and Systems Modeling 6 (2) (2007) 139–162.
[45] A. Tang, A. Nicholson, Y. Jin, J. Han, Using Bayesian belief networks for change impact analysis in architecture design, Journal of Systems and Software 80 (1) (2007) 1–148.
[46] O.M.G. UML, 2.0 Superstructure Specification, OMG ed., 2003.
[47] P. Van Gorp, D. Janssens, T. Gardner, Write once, deploy N: a performance oriented MDA case study, in: Proceedings of the IEEE International Conference on Enterprise Distributed Object Computing, pp. 123–134, 2004.
[48] A. von Knethen, M. Grund, QuaTrace: A tool environment for (semi-)automatic impact analysis based on traces, in: Proceedings of the International Conference on Software Maintenance, pp. 246–255, 2003.
[49] Z. Xing, E. Stroulia, Differencing logical UML models, Automated software engineering, in: Special Issue on selected papers from the 20th International Conference on Automated Software Engineering (ASE'2005) vol. 14, pp. 127–259, 2005.
[50] Z. Xing, E. Stroulia, Refactoring Detection based on UMLDiff change-facts queries, in: Proceedings of the The 13th Working Conference on Reverse Engineering (WCRE'06), 2006.