2017 IEEE/ACM 39th International Conference on Software Engineering

UML Diagram Refinement (focusing on class- and use case diagrams)

David Faitelson
Software Engineering Department
Afeka Tel Aviv Academic College of Engineering, Israel
Email: davidf@afeka.ac.il

Shmuel Tyszberowicz
School of Computer Science
Academic College of Tel Aviv-Yaffo, Israel
Email: tyshbe@mta.ac.il

***Abstract*—Large and complicated UML models are not useful, because they are difficult to understand. This problem can be solved by using several diagrams of the same system at different levels of abstraction. Unfortunately, UML does not define an explicit set of rules for ensuring that diagrams at different levels of abstraction are consistent. We define such a set of rules, that we call *diagram refinement*. Diagram refinement is intuitive, and applicable to several kinds of UML diagrams (mostly to structural diagrams but also to use case diagrams), yet it rests on a solid mathematical basis—the theory of graph homomorphisms. We illustrate its usefulness with a series of examples.**

***Keywords*-Refinement; UML; Class diagram; Use case diagram; Graph homomorphism; Design patterns**

I. Introduction

Large and complicated UML models are not useful because they are very difficult to understand. Unfortunately, models of complex systems are often complex as well [1]. The traditional advice for dealing with this problem is to group related entities into packages [2]. But because packages are only organizational units that do not represent any entity in the domain, diagrams that consist only of packages convey very little information about the domain we are trying to model. For example, suppose that we partition a large class diagram into several packages, but there are important associations between classes in different packages. We cannot understand the structure of the system just by looking at the relationships between the packages. We must open the packages to look at the classes inside. But then it is not clear what we have gained by the packages. We believe that what is missing here is explicit support for relating diagrams that model the same system at different levels of abstraction.

The UML offers a relationship for asserting that one model is more abstract (or refined) than another model [3]. But it does not offer any set of rules for determining if such an assertion is indeed correct. Unfortunately, without such rules it is very difficult to place our confidence in the relationship. For example, it is difficult to discuss an abstract diagram with the customer if we are not certain that it is consistent with the more refined version that the developers use.

There have been earlier attempts to add a theory of refinement to UML models (e.g., [4], [5], [6], [7], [8], [9]). However, to the best of our knowledge, all this previous work demands that the engineers will use a formal framework (as in a methods like Z [10], B [11], or Object-Z [9]) and reason about their models in terms of a mathematical theory (e.g., the theory of data refinement [10], description logic [5], or category theory [7]). Unfortunately, the level of mathematical sophistication required in order to use these approaches is well beyond what most engineers are willing to accept.

Our contribution is a simple set of rules, that we call diagram refinement, for checking if a concrete diagram is consistent with (that is, refines) a more abstract diagram. Our rules have a sound mathematical basis, yet they do not require knowledge of this basis in order to be applied. In fact, we believe that diagram refinement has a simple intuitive meaning that can be easily understood by any engineer who is familiar with UML models. In addition, our rules are based on a simple relational semantics. As a result, we can apply the rules to any kind of diagram for which we can define such semantics. In this paper we describe a relational semantics for two popular UML diagrams: class diagrams and use case diagrams.[1]

Diagram refinement can be used in a top down fashion, to develop new software by gradually enriching abstract models with more details, while ensuring that the detailed description preserves the original abstraction. This is important if we are to use the abstract models to explain and document the system. Alternatively, diagram refinement can be used in a bottom up fashion, to reverse engineer existing systems in order to discover the abstractions encoded in the concrete models.

Certainly, having a precise notion of refinement is essential for tool builders. For example, it may be used to ensure that when we move from an analysis viewpoint to a design viewpoint, the more detailed design diagram is consistent with the analysis diagram. But even practitioners that use diagrams only as sketches may find the theory useful. If you know the theory you can make informed deviations which you understand, and are under your control. In this case your abstractions will still be useful. However, without a theory there is no way to know what is the connection between the abstract and concrete models. In this case, you better throw away the abstraction, as using it would lead your readers astray, and probably do more harm than good.

In the next section we explain the rules of class diagram refinement. We then extend the rules to deal with association

[1] According to [12] class diagrams are the most widely used UML diagrams.

1558-1225/17 $31.00 © 2017 IEEE
DOI 10.1109/ICSE.2017.73

multiplicities, and demonstrate (in Sect. III) their usefulness by an analysis of a large and complicated class diagram, taken from a real world project. In Sect. IV we provide a mathematical foundation for the rules, where we show that diagrams induce particular kinds of graphs, and that the rules of valid refinement are equivalent to graph homomorphisms. Next, in Sect. V, we show that diagram refinement can be used to formally capture the use of design patterns and to clarify use case diagrams. We describe the relational semantics that underlies diagram refinement in Sect. VI. Related work is presented in Sect. VII, and we conclude in Sect. VIII.

II. Class diagram refinement

Lack of abstraction is a problem of large diagrams, but it also exists in small diagrams, and they are better tools for illustrating our ideas. Consider the simple diagram in the left corner of Fig. 1. We have a class *Person* with an association that represents the relationship between a person's parents and children.[2] Suppose we want to assert that a person is either a male or a female, the male is the father and the female is the mother. How can we represent this information in the model? A naive approach will add this information to the original diagram. Unfortunately, the result is inconsistent with the situation we are trying to model. It now appears that each person has two parents and in addition has a male father and a female mother. A better approach is to declare that *parents* is a derived association of *mother* and *father*. But this only adds another level of complexity to the diagram. When the diagram is large and complicated, declaring that some associations are derived from others only makes it more difficult to understand.

The key to solving this problem is to observe that we are describing the same structure at two different levels of abstraction. At a high level, we are not interested in the gender of the parents, so we are happy to model the situation as a single association between persons. At the more detailed level, we refine our model to distinguish the gender of the parents, and as a result create a more elaborate model. By separating the different levels of abstraction into separate diagrams, we facilitate modular development. We can reason about key properties at the abstract level, and later focus on the details, without hiding the abstraction. Of course, we must ensure that the added details are consistent with the abstraction, see Fig. 1 (note that when the multiplicity is 1 we omit it). But simply stating that one diagram refines another is not sufficient, because we run the risk of creating concrete models that are not consistent with their abstract versions. Therefore, in the following sections we define an explicit set of rules for determining when one diagram refines another.

A. Valid Diagram Refinement

A diagram refinement is a mapping from an abstract diagram to a concrete diagram. To avoid cluttering refinements with trivial information, we adopt the convention that if an entity in the abstract diagram does not appear in the mapping, then it is mapped to the entity with the same name in the concrete diagram (of course we have to check that such an entity exists). For example, the mapping

$$\begin{aligned} \mathit{father} &\rightarrow \mathit{parents} \\ \mathit{mother} &\rightarrow \mathit{parents} \end{aligned} \tag{1}$$

determines the refinement between the diagrams in Fig. 1. The mapping states that *Person* in the concrete diagram represents *Person* in the abstract diagram, and that the associations *father* and *mother*, correspond to *parents*. We did not map *Male* and *Female* to *Person* as these classes are new additions that do not represent any class in the abstract diagram. Diagram refinement is a relation between two diagrams, not between individual entities.

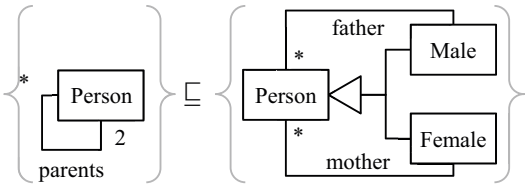


Fig. 1. An abstract diagram (on the left) and its refinement (right). The class *Person* in the concrete diagram represents the class *Person* in the abstract diagram. The *father* and *mother* associations represent the *parents* association.

For a diagram refinement to be valid, these rules must hold:

1) Every class in the abstract diagram must map to exactly one class in the concrete diagram.
2) Every association in the abstract diagram must map to at least one association in the concrete diagram.
3) If association c in the concrete diagram maps to association a in the abstract diagram, c's classifiers must map to a's classifiers or to subtypes of these classifiers.
4) If class x is a subtype of class y in the abstract diagram, then the class that is mapped to x in the concrete diagram must be a subtype of the class that maps to y.
5) If class x is a subtype of class y in the concrete diagram, and they are mapped to classes a and correspondingly b in the abstract diagram, then a must be a subtype of b.

Rules 1 and 2 ensure that the concrete diagram represents at least all the information in the abstract diagram, whereas rules 3-5 ensure that the structure of the concrete diagram is consistent with the structure of the abstract diagram. Rule 3 ensures that every association between two concepts at the abstract level, remains between the same two concepts at the concrete level. Rule 4 ensures that refinement maps subtypes to subtypes. Rule 5 ensures that any subtype relationship that appears in the concrete diagram and can be represented in the abstract diagram, will appear in the abstract diagram.

By using these rules we can catch invalid refinement attempts. For example, the diagram in Fig. 2 is not a valid refinement of the abstract diagram in Fig. 1, as we cannot map the associations *father* and *mother* to *parents* without

[2] We are aware that it is common to separately name each end of the association, but using a single name for the association simplifies the presentation. We later describe how to apply the ideas to associations with separate names.

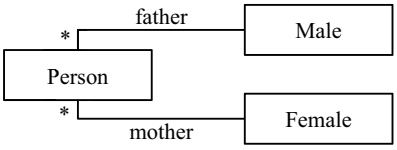


Fig. 2. An invalid refinement of the abstract person diagram, violating rule 3.

violating rule 3. (It also violates rule 4 because every class is a subtype of itself.) To fix this refinement we must make *Male* and *Female* subtypes of *Person*. Then the source and target of *father* and *mother* are both subtypes of *Person*, and we can map them to *parents* without violating any of the rules. An important property that these rules ensure is that any composition of abstract associations is mapped to some composition of concrete associations. Therefore, a concrete diagram can simulate (in greater details) any path of association traversals. For example, in the abstract diagram in Fig. 1 it is possible to traverse the *parents* association as many times as we like, producing ever distant ancestors. Similarly, in the refined diagram it is possible to follow the same path using *father* and *mother*, thus not only identifying the ancestors, but also their gender.

B. Refinement with Multiplicity Constraints

So far we have ignored the multiplicity constraints on the associations, treating each association as an unrestricted relation. In this section we model the meaning of the multiplicity constraints in terms of constraints on relations.[3] This makes it possible to capture refinement relationships between diagrams that contain associations of different multiplicities.

We will write $(x - y)$ to denote an association with multiplicity constraints x and y, where x and y can be either a single symbol: 1, ? (we use ? to denote the optional association 0..1), $*$, or a range of the form $m..n$, where $m \leq n$ are natural numbers. For example, $(* - *)$ represents a many-to-many association. When we do not care about the cardinality of one end, we replace it with a variable. For instance, $(x{-}?)$ means either $(*{-}?)$ or $(i..j{-}?)$. The meaning of an association with multiplicity constraints follows from the implications of the constraints to the relation that we identify with the association: unconstrained associations (of the form $(* - *)$) correspond to unconstrained relations; associations where one end is constrained to be either mandatory or optional correspond to functions (total or partial). In the general case, an association x between classes A and B with multiplicity constraints $(i..j - m..n)$, corresponds to the constraints

$$\forall a : A \mid m \leq \#a.x \leq n$$
$$\forall b : B \mid i \leq \#b.{}^{\sim}x \leq j$$

where $\#s$ is the cardinality of the set s, and ${}^{\sim}x$ is the association x seen from B's side (which corresponds to the inverse relation of x under a relational interpretation of associations).

[3]We rely on a relational semantics of class diagrams explained in Sect. VI.

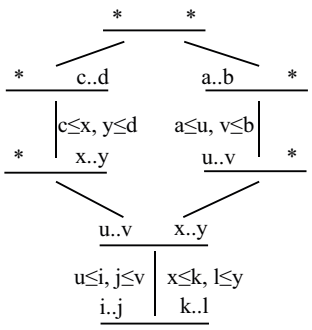


Fig. 3. A lattice of multiplicity constraints. The lattice supports all possible multiplicities, by plugging the appropriate values. For example, a multiplicity of 1 corresponds to 1..1, an optional multiplicity to 0..1, etc. The edges that are annotated with an inequality exist only when the inequalities hold.

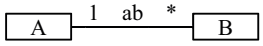


Fig. 4. An abstract association.

Using this interpretation, we build a lattice of association types that defines a partial order on the different association multiplicities[4]; see Fig. 3. The lattice defines the subset relations between the different kinds of associations. If c appears in the lattice as a child of p, then every association of type c is a subset of p. For instance, if an association is a surjective injection $(? - 1)$ then it is also an injection $(?{-}?)$. In a valid refinement, the associations of the concrete diagram must be either equal to, or less than, the associations of the abstract relation, where 'less than' is defined by the lattice. In addition, to take care of the subtype relations between the classes, we add the following constraints:

- Functionality: when several associations refine a functional ($(x{-}?)$ or $(x - 1)$) association, their union must be functional as well.
- Onto: When several associations refine an $(x - 1)$ association, the union of their sources must be equal to the source of the abstract association that they refine.

For example, Fig. 4 shows an abstract model with a single one-to-many association. Figure 5 shows two valid refinements of the abstract model, and Fig. 6 shows three invalid refinements.

C. Refinement with Other Constraints

We now explain how we can extend our notion of diagram refinement to additional UML class diagram constructs.

1) Aggregation and Composition: Aggregation and (particularly) composition, constrain the model in terms of how a particular object is shared between other objects. This type of constraint is mostly independent from multiplicity constraints. Therefore we consider the different aggregation kinds separately from the multiplicity of the association. Any association end in the abstract diagram may be mapped to

[4]To simplify the lattice's diagram, we use the fact that the general notation $(i..j - m..n)$ can describe the optional and mandatory multiplicities: A mandatory end corresponds to $n..n$, and an optional end corresponds to 0..1.

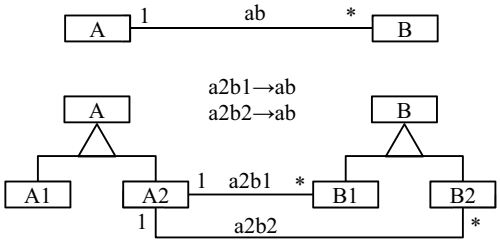


Fig. 5. Two valid refinements of Fig 4. Note that in the lower diagram we satisfy the onto constraint because the union of the right hand side of $a2b1$'s and $a2b2$'s source ($B1 \cup B2$) is equal to B.

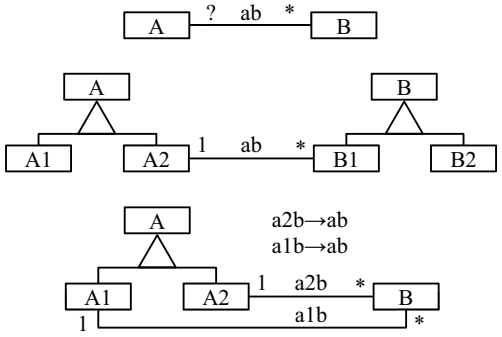


Fig. 6. Invalid refinements of the abstract diagram in Fig. 4. From top to bottom: the first is invalid as it violates the order determined by the lattice; the second is invalid as it violates the onto constraint (only $B2$s are mapped to As, which fails to ensure that every B has an A); the third is invalid because it violates the functionality constraint (the same B is mapped to different As).

either a shared (aggregation) or a composite (composition) aggregation in the concrete diagram. A shared kind in the abstract diagram may be mapped to a composite kind in the concrete diagram. This reflects the fact that the constraints introduced by aggregation are weaker than those of composition. Every configuration of objects that satisfies the constraints of composition, also satisfies the constraints of aggregation. Thus, replacing aggregation with composition results in a more specific but still consistent, and therefore refined, model.

2) Qualified associations: Qualifying an association reduces the multiplicity of the original (unqualified) association. Therefore, it is valid to map an association a in the abstract diagram to a qualified association c in the concrete diagram, provided that $c \leq a$ (as given by the lattice). This is possible even when a is already qualified. See Fig. 7 for an example.

3) OCL constraints: OCL is used to specify constraints that are difficult or impossible to express with diagrammatic notation alone. If we think of a class diagram as a notation for specifying the structural relationships between objects, then adding OCL constraints further reduces the set of valid object structures, rejecting all the structures that fail to satisfy the constraints. For example, we may constrain the diagram in Fig. 1 by insisting that a person is never its own parent (we use OCL packages to separate the different diagrams):

```
package Abstract
```

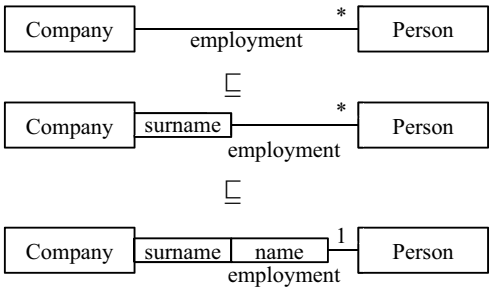


Fig. 7. A simple model with two levels of refinement. In the first refinement we qualify the employment relation on the surname attribute of person, but there could still be people with the same surname in the company. In the second refinement we qualify the qualified association.

```
context Person inv:
  self.parents->forAll(p| p <> self)
endpackage
```

We may also constraint the concrete diagram in the same spirit:

```
package Concrete
context Person inv:
  self.mother->forAll(m | m <> self) and
  self.father->forAll(f | f <> self)
endpackage
```

For a concrete diagram to refine (be consistent with) an abstract diagram with OCL constraints, we must require that all the constraints in the abstract diagram be implied by the constraints of the concrete diagram, with perhaps additional constraints added to the concrete diagram. However, a concrete diagram often encodes the associations in the abstract diagram using different, more concrete, associations. Therefore we must use the refinement mapping to translate the meaning of the constraints in the abstract diagram to the structure of the concrete diagram. In our example, to keep the refinement in Fig. 1 valid, we must ensure that no person is a parent of itself. However, the concrete diagram does not have a *parents* association; instead, it represents this abstract association using *mother* and *father*.

Taking a page from the theory of data refinement [10], we may represent the refinement mapping as a predicate, *retrieve*, that explicitly describes how the concrete model encodes the information in the abstract model. Unfortunately, it appears that it is impossible to write OCL expressions that link together different diagrams. It appears that OCL expressions are intended to accompany an individual UML model. As far as we could see, there is no way to say, e.g., that a class *Person* in one diagram is the same (represents the same set of objects) as a class *Person* in a different diagram.

Nevertheless, we think it is worthwhile to illustrate how we can describe diagram refinement in OCL, under the assumption that it is possible to relate entities in different diagrams. Specifically, we assume that it is possible to assert that two class names in different diagrams represent the same class. Figure 8 shows the retrieve predicate for our example.

```
context Person def :
retrieve() : Boolean =
 self.parents.forAll(p |
  self.mother.includes(p) or
  self.father.includes(p))
 self.mother.
  forAll(p|self.parents.includes(p))
 self.father.
  forAll(p|self.parents.includes(p))
```

Fig. 8. An OCL expression that describes how the concrete diagram encodes the structure of the abstract diagram. We assume that *Person* represents the same set of objects in both diagrams, and check that the *parents* association in the abstract diagram is the union of the *mother* and *father* associations in the concrete diagram.

We now specify when diagram refinement is valid. Assume *Abstract* :: *Inv*() is an OCL boolean expression that represents the constraints in the abstract diagram, *Concrete* :: *Inv* represents the constraints in the concrete diagram, and *R* represents the retrieve relation. Then the refinement is valid precisely when *R* is a valid diagram refinement and the following OCL expression is true:

```
Concrete::Inv() and R()
  implies Abstract::Inv()
```

III. An Extended Example

To illustrate the utility of diagram refinement, we have applied it to a diagram that models the conceptual entities of AstroGrid, a virtual astronomical observatory (see `http://wiki.astrogrid.org`). It provides a registry for locating astronomical data, observation platforms, and computational services to access and analyze the data in a uniform way.

The AstroGrid diagram consists of 60 classes and 74 relationships (counting associations and subtypes relations together). Unfortunately, it is too big to fit in a single page and still be readable. To get a feeling for its complexity, we have replaced each class in the diagram with a point, and each association with an edge; see Fig. 9. We kept intact only the note that accompanied the original diagram. The complete diagram is available (with the permission of its author) online [13]. We believe that this diagram captures the dilemma that faces the modeler of a complex domain. On the one hand it is too big to be useful (and indeed, it was abandoned for this fact), yet, on the other hand, it is still missing many important problem domain entities. Clearly, it is impossible to satisfy in a single diagram both the demand for faithful and detailed representation of the conceptual domain and the demand for simplicity and clarity that makes such diagrams useful.

In the following paragraphs we describe our experience of using diagram refinement to better structure the large AstroGrid UML class diagram. As we were not involved in the design of this system, we did not have a perfect understanding of all its details. Therefore, as we made progress in trying

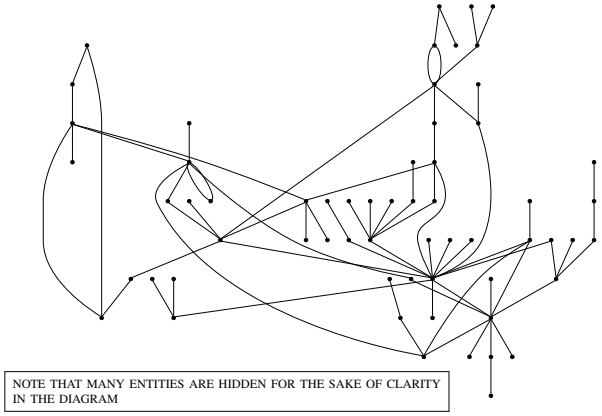


Fig. 9. An abstraction of the entire AstroGrid UML class diagram. Each point is a class and each edge is a relationship (association or subtype) between the classes. The note appeared in the original diagram.

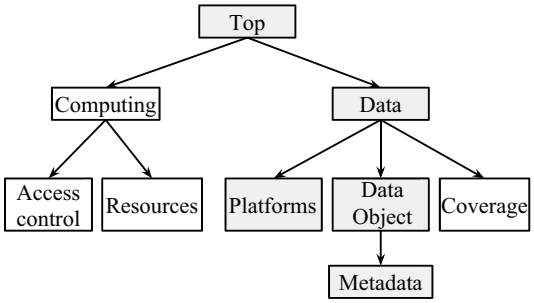


Fig. 10. The refinement tree, showing all the refinements we have performed on the various areas of the AtroGrid model. The names on the nodes are the names of the diagrams that focus on that particular subject. The top most diagram is the most abstract view of the system. If we combine all the leaves into a single diagram, we get the original AstroGrid diagram. The shaded diagrams appear in the paper, the rest are available at http://tinyurl.com/jz4akby.

to abstract the original diagram, we made mistakes which diagram refinement helped us to find and correct. We know from experience that developing a system model is a similar process. We make mistakes, correct the model, and repeat, until we do not find any more mistakes. Any method that can help us catch mistakes is thus useful. The following report illustrates how diagram refinement may be useful in the context of the development process. After several sessions of studying the original diagram and any AstroGrid design documents we could find online, we felt that we had understood it well enough to be able to create an abstraction/refinement tree of diagrams that describes the essential concepts of the problem domain at various levels of abstraction; the result is shown in Fig. 10. Figure 11 shows the diagram Top, which is the most abstract view of the system.

The first refinement that we have explored is to elaborate the relationships between the astronomical data and the observing platforms that produce this data. We can see this refinement in Fig. 12. Observation platforms hold instruments that perform observations and produce data objects. Each observing

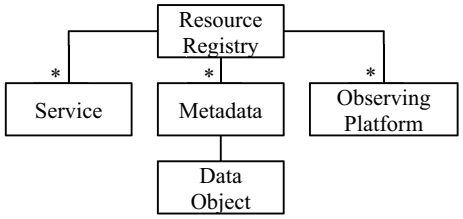


Fig. 11. The diagram Top, representing the most abstract view of the AstroGrid system. We can see that the system has a registry through which we can locate services, metadata that holds the storage location of data objects, and information about observing platforms.

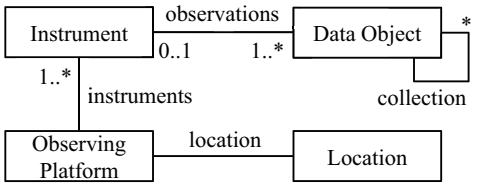


Fig. 12. The diagram Data, revealing one level of details of the structure of the astronomical data and the entities that collect this data. Data is created by instruments that are located on observing platforms.

platform has a physical location. In addition, the system keeps track of the location in which the observations are stored. Our next refinement was to reveal the hierarchical structure of the data collected by the system. In the refined model the data object entity becomes a super type of three different kinds of entities: physical constants, observations, and data sets. Only data sets may hold collections of data objects; see Fig. 13.

We have then turned our attention to the structure of the observation platforms. We have learned from the original diagram that an observing platform can be either a stationary observatory or a spacecraft. Spacecrafts have an orbit that, we believed, represents their location; we show this refinement in Fig. 14. But it turned out that we had a mistake. The problem is with the multiplicities of the two associations. The *location* is an 1-to-1 association, and so by the *onto* constraint (Sect. II-B) the source of *orbit* must be equal to *ObservingPlatform*. However, the source of *orbit* is *SpaceCraft*, which leaves

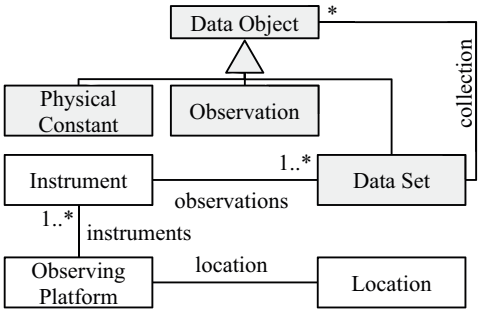


Fig. 13. The diagram Data Object, elaborating the hierarchical structure of data objects. Note that one end of *observations* is now mandatory, whereas in the abstract diagram it was optional.

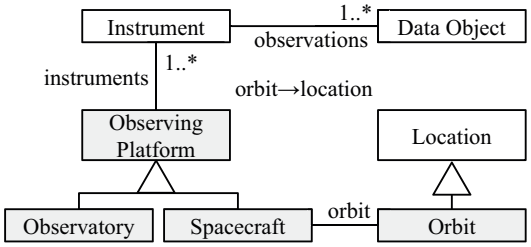


Fig. 14. The diagram Platforms, a different refinement of the diagram Data in Fig. 12, revealing the structure of the observing platforms. The refinement is wrong as it maps *orbit* to *location*, leaving *Observatory* without a location.

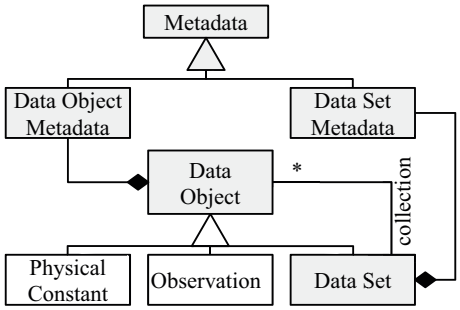


Fig. 15. The diagram Metadata, elaborating the relationships between the structure of data objects and their meta data objects.

Observatory without a location. This analysis has led us to the conclusion that *orbit* is not a concrete representation of *location*, but rather an additional kind of location that is unique to *SpaceCraft* entities. The correct refinement appears online.

Another interesting refinement concerns the relationship between the data objects and their meta data. In AstroGrid, the meta data objects record where the data objects (observation measurements) are being stored. At the abstract level (Fig. 11) we simply associate each data object with its meta data object. However, as we can see in the original diagram, meta data for data sets is different from meta data for data objects. We present this state of affairs in Fig. 15. Interestingly, even though this is a valid refinement, it appears there is a mistake in the original diagram. Because data sets are also data objects, they now have two meta data objects: one that they get by inheriting from data object and one that they get directly. For lack of space we do not describe all the refinements in the refinement tree, and we show only the interesting parts of the diagrams. The full diagrams are available on the website.

If we combine the diagrams in the leaves into a single diagram we get back the original diagram. Nevertheless, it is much more convenient to first select a particular topic from the tree and then look at the relevant diagram that has about 10 classes, rather than struggling to locate the relevant information in a diagram that consists of about 60 classes.

To conclude, if we had just tried to abstract the model without the assistance of diagram refinement, we would have probably not detected the mistake that we have just described:

it is subtle and requires careful attention, and among all the details in the model there was nothing a-priori that could have directed us to suspect these particular abstractions.

IV. A Mathematical Foundation

In this section we show that class diagrams naturally translate into graphs and that a valid refinement translates into a graph homomorphism. As a result we can be certain that our definition of refinement is internally consistent.[5] But first, some preparatory definitions. A class diagram D is a tuple $D = (D_0, D_1, \leq)$, where D_0 is the set of classes in the diagram, D_1 is the set of associations in the diagram, and $\leq$ is the transitive closure of the subtype relation between the classes in the diagram (we demand that $\leq$ is a partial order). We will write $a : x \to y$ to assert that the association a is an association between the class x and the class y. In that case we may also write $source(a) = x$ and $target(a) = y$.[6] Finally, we will assume that every class diagram has an implicit class $\top$ that is the supertype of all the classes in the diagram. This can always be achieved in our mathematical framework, and it will simplify the mathematics.

Diagram refinement R between a concrete diagram C and an abstract diagram A, is a pair of functions (R_0, R_1), where $R_0 : C_0 \to A_0$ maps classes in the concrete diagram (C_0) to classes in the abstract diagram (A_0), and $R_1 : C_1 \to A_1$ maps associations in the concrete diagram (C_1) to associations in the abstract diagram (A_1). The refinement is valid exactly when

B1. R_0 is a partial bijection (injective and surjective),

B2. R_1 is surjective,

B3. $\forall c : C_1, a : A_1 \mid R_1(c) = a \Rightarrow$
$R_0(source(c)) \leq R_0(source(a))$
$\wedge\, R_0(target(c)) \leq R_0(target(a))$,

B4. $\forall a_1, a_2 : A_1 \mid a_1 \leq a_2 \Leftrightarrow R_0^{-1}(a_1) \leq R_0^{-1}(a_2)$.

These rules are a formal translation of the informal rules that we have given in Sect. II-A, except that the two implications in the final equivalence represent rules 4 and 5 together.

From conditions (B1) and (B4) above we see that in a valid refinement, R_0 is injective, and monotonic with respect to the subtype relation. Unfortunately R_0 is not total, and this may complicate the mathematics that will follow. It will therefore be useful to define a total function that will complete whatever is missing in R_0. Let us call this function ρ. We define ρ as

$$\rho(c) = R_0(\min\{\, c' : \operatorname{dom} R_0 \mid c \leq c' \,\}),$$

where by $\min(s)$ we mean the class in s that is a subtype of all other classes in s, or the special class $\top$ if no such class exists. (We assume that R_0 maps $\top$ to $\top$.) Intuitively, ρ traverses a path from c upwards through the subtype hierarchy, looking for the first supertype of c that R_0 maps to a class in the abstract diagram, and maps c to that class; see Fig. 16.

A useful property of ρ is that when R is valid, then for any concrete association α that R_1 maps,

$$R_1(\alpha : x \to y) : \rho(x) \to \rho(y). \tag{2}$$

[5]We plan to consider multiplicity constraints in future work.

[6]This notation facilitates referencing the ends of associations. It does not define a direction.

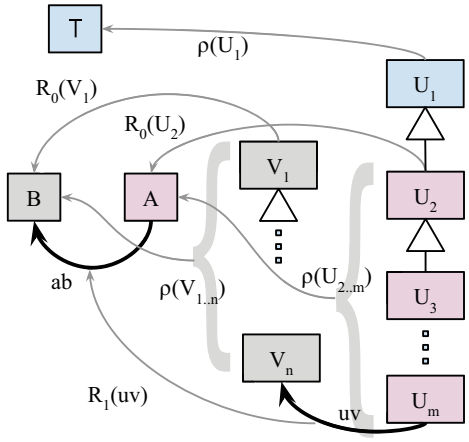


Fig. 16. The structure of a valid refinement. R maps the association uv to ab even though U_m and V_n are not mapped. But as V_1 is the least supertype of V_n that is mapped by R (and similarly for U_m), we have $R_1(uv) : \rho(source(uv)) \to \rho(target(uv))$.

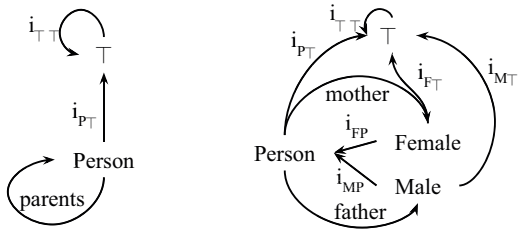


Fig. 17. The graphs that correspond to the class diagrams in Fig. 1. To avoid cluttering the graphs, we did not draw the future edges (there is a future edge between any two vertices in each graph) and the inverse association edges.

1) Diagrams are Graphs: Given a class diagram D, we construct a graph $G(D)$ whose vertices are D's classes, and whose edges correspond to associations between the classes of D. For each association a between A and B we define two edges, $a : A \to B$ and $\sim a : B \to A$. This way we capture the fact that in UML an association has two ends.[7] In addition, whenever class B is an immediate subtype of class A, we add to the graph an *inclusion* edge $i_{BA} : B \to A$. The edge i_{BA} represents a total injection from the instances of B to the instances of A, making the subtype relation between the two classes explicit. Next, for each pair of vertices x and y in the graph we add an edge $future_{xy} : x \to y$. This edge represents any associations between x and y that future refinements may create. Finally, we add to the graph the vertex $\top$, and add an edge $i_{x\top} : x \to \top$ for every vertex x in the graph. The vertex $\top$ represents the supertype of all the classes in the diagram (much like the class *Object* is the base class of all the classes in Java). The reason for this particular construction will become apparent in the next section. For example, the two diagrams we used in Fig. 1 correspond to the two graphs shown in Fig. 17.

[7]If our diagram has associations with different names for each role, we give each edge in the pair the name of the corresponding role.

2) Valid Refinement = Homomorphism: A graph homomorphism H from graph G to graph G' is a pair of functions, $H = (H_0, H_1)$, where the first, H_0, takes the vertices of G to those of G', and the second, H_1, takes the edges of G to the edges of G'. In addition, this pair of functions must satisfy the property that if $u : m \to n$ is an edge in G, then $H_1(u) : H_0(m) \to H_0(n)$ is an edge in G'.

Given a refinement $R = (R_0, R_1)$, we define a function pair $H(R) = (H_0, H_1)$, according to the following construction:

C1. $H_0 = \rho$,
C2. if $R_1(a) = a'$ then $H_1(a) = a'$ and $H_1(\sim a) = \sim a'$,
C3. if $a : x \to y \notin \mathrm{dom}\, R_1$ then $H_1(a) = \mathit{future} : H_0(x) \to H_0(y)$ and $H_1(\sim a) = \mathit{future} : H_0(y) \to H_0(x)$,
C4. $H_1(i_{xy}) = i_{H_0(x)H_0(y)}$,
C5. $H_1(\mathit{future}_{xy}) = \mathit{future}_{H_0(x)H_0(y)}$.

We now prove the following theorem:

Theorem 1: R is a valid diagram refinement from diagram C to diagram A iff $H(R)$ is a graph homomorphism that maps all vertices, associations edges, and inclusion edges of $G(A)$.

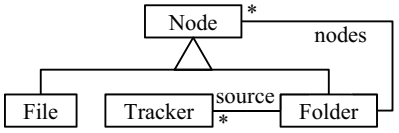
Proof: Assume that R is a valid diagram refinement. Let $a : x \to y$ be an edge in $G(C)$. It is either an inclusion edge, or a future edge, or an association edge. In the first case, we know that the inclusion edge $i_{H_0(x)H_0(y)}$ exists in $G(A)$, as valid refinement maps subtypes to subtypes (by rule B4). In the second case we know that $\mathit{future}_{H_0(x)H_0(y)}$ exists in $G(A)$ since by construction there is a future edge between any two vertices in $G(A)$. In the last case, if R does not map a, then H_1 maps a to $\mathit{future} : H_0(x) \to H_0(y)$ by rule C5. Finally, if R does map a, then by rule B2 and Eq. 2 we have $H_1(a) = R_1(a) : \rho(x) \to \rho(y)$, and from rule B1 we get that $H_1(a) : H_0(x) \to H_0(y)$. Therefore, if R is a valid diagram refinement then $H(R)$ is a graph homomorphism.

Now assume that R is not valid. If R violates rules B1, or B2, or B3, then $H(R)$ will not map all the vertices, or all the association edges, or all the subtype edges of $G(A)$. If it violates rule B4, then $H(R)$ will not be a homomorphism. ■

V. Additional applications

We now show additional applications of diagram refinement. First we show that it is possible to use diagram refinement to formally capture design patterns in a class diagram, and then we show that it is possible to use diagram refinement to improve the structure of use case diagrams (UCDs).

1) Documenting Design Patterns: Refinement of class diagrams can explicitly capture design patterns in a concrete design. For example, Fig. 18 (top) is a class diagram of a simple file system. This is a clear example of the composite design pattern, yet it has no formal relationship to thispattern diagram. This is not a problem when the diagram is simple, but when the class diagram consists of many nodes and associations, it becomes difficult to identify the patterns, especially as in many cases the same classes participate in several patterns. For example, a folder may also act as a subject in the observer pattern, notifying interested observers about changes to the files that it contains. By using class diagram refinement we can explicitly capture the relationship between design patterns and their manifestation in our particular design (Fig. 18, bottom).



Refinement of Composite	Refinement of Observer
Node → *Component*	*Folder* → *Subject*
File → *Leaf*	*Tracker* → *Observer*
Folder → *Composite*	*source* → *subject*
nodes → *children*	

Fig. 18. A class diagram that refines two patterns. We show two refinement mappings, to the composite and to observer design patterns. We did not include the diagrams of these two well known design patterns to save space.

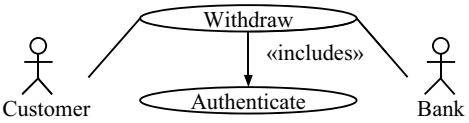


Fig. 19. A simple UCD of a customer withdrawing cash from an ATM. Here it is not clear that the bank is not involved in the card authentication step.

2) Refinement of Use Case Diagrams: UCDs depict two separate concepts: i) the relationships between actors and the use cases in which they participate, and ii) the structure of the use cases. This can often cause problems. Consider, e.g., the use case diagram in Fig. 19. When a customer withdraws cash, she first has to authenticate her card, but this does not involve the bank. The bank is only used later to authorize the cash withdrawal transaction and deduct the money. How can we express this in the diagram? Currently the diagram simply depicts the customer and the bank as participating in the use case. Should we remove the link between the customer and *Withdraw* and instead link it to *Authenticate*?

But this misses the fact that the customer participates in the entire use case. Should we just add a new link from *Customer* to *Authenticate*? This is acceptable, perhaps, when the diagram does not contain a lot of *include* relationships; otherwise, it could quickly become flooded with arrows, hindering our ability to understand it. As before, this is a case of looking at the same situation from different abstraction levels. At the abstract level we are not interested in the internal composition of the use case, thus we would draw it as in the upper part of Fig. 20. At the more concrete level we describe the internal structure of the use case, and then show which parts are relevant to which actors, as in the lower part of the figure.

We translate a use case diagram into a graph in a similar way to how we translate a class diagram into a graph, except that instead of the subtype relation we have *include* and *extend* relations. As in the case of class diagram refinement, the rules of use case diagram refinement are that whenever an edge e in the concrete diagram corresponds to an edge e' in the

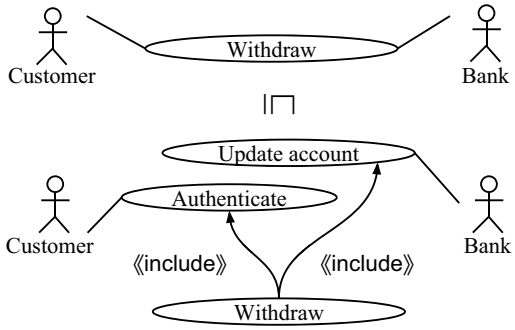


Fig. 20. An abstract use case and its refinement. At an abstract level the *Withdraw* use case has no structure. At the concrete level we see its structure and can refine the arrows to indicate which parts are used by which actor.

abstract diagram, the source and target of e must correspond to the source and target of e'. For example, the edge adjacent to *Customer* in the concrete use case corresponds to the edge between *Customer* and *Withdraw* in the abstract use case, since *Authenticate* is included by *Withdraw* which corresponds to *Withdraw* in the abstract diagram.

VI. Semantics

The semantics that underlies diagram refinement is best understood in terms of a simple relational model that consists of a finite collection of sets (the sets may be infinite) and relations between these sets. When we refine an abstract collection of sets and relations, the result is a new, concrete, collection of sets and relations, from which we can assemble the original abstract collection. To be more precise, if a set A appears in the abstract collection, then it should also appear in the concrete collection. If a relation $r : A \leftrightarrow A'$ appears in the abstract collection, then it must appear in the concrete collection, but it may be represented in the concrete collection by a set of relations $x_i : A_i \leftrightarrow A'_i$, provided that A_i is a subtype of A and that A'_i is a subtype of A'. This ensures that when we move from the abstract to the concrete representation, we do not change the meaning of the relations.

We have defined our refinement rules precisely to ensure that this notion of consistency between the abstract and concrete diagrams is preserved. Given any kind of diagram, if we can provide it with such a relational semantics, we may use our refinement rules to support an incremental and consistent development of these diagrams.

We now demonstrate such a relational semantics for two UML diagrams. These examples may also serve as models for the development of semantics for additional kinds of diagrams.

1) Class Diagram Semantics: We use a simple relational semantics for class diagrams. We consider each class to represent the set of all its instances, and interpret the subtype relationship as a subset relation.[8] In addition, we interpret binary associations as binary relations. As a result we can use relational composition to model navigation. For example, when we navigate from parent to itself twice, we get the grandparent, this corresponds to the relational composition $parent \circ parent$. Finally, we interpret attributes as functional relations. An attribute a of class A whose type is T is a function a from A to T. This way we do not have to make a special case for the semantics of attributes. Figure 21 shows (on the left) the relational semantics of the class diagram in Fig. 1.

[8]Unless otherwise stated, we interpret subtype as disjoint and complete.

$Person = Male \cup Female$	$Authenticate \subseteq Withdraw$
$Male \cap Female = \emptyset$	$Update\ account \subseteq Withdraw$
$father \subseteq Person \times Male$	$Customer \in Authenticate$
$mother \subseteq Person \times Female$	$Bank \in Update\ account$

Fig. 21. On the left, the semantics of the class diagram in Fig. 1. On the right, the semantics of the use case diagram in Fig. 20.

2) Use Case Diagram Semantics: We consider each use case in the diagram to represent the set of all actors that participate in the use case. When actor a is associated to use case U it means that a is a member of U. We consider the *include* and *extend* relationships as subset relations. When use case A includes use case B, it means that B is a subset of A; when A extends B, it means that B is a subset of A. We illustrate these ideas in Fig. 21, where we show (on the right) the semantics of the use case diagram in Fig. 20.

VII. Related work

Hnatkowska et al. [14] define abstract laws for refinement of UML models. They represent UML models as sets of terms and rules, and establish laws for determining when a transformation of terms and rules is a valid refinement. The discussion is very abstract, and cannot be used as a tool by engineers. In addition, they insist that an abstract term must be refined by exactly one concrete term, but we believe this is too strong. For example, it would rule out using *mother* and *father* to represent *parent*.

Evans [15] uses a formal semantics for class diagrams to define transformation rules for deducing an abstract class diagram from a more concrete one. We have found that all the valid deductions in [15] are also valid diagram refinements, and all the invalid deductions are invalid diagram refinements.

Bergner et al. [16] discuss the importance of refinement for the development of class diagrams. This work, however, defines refinement to mean any kind of transformation between development levels. It focuses on documenting the transformations, but offers no way to check consistency.

Ammar et al. [17] investigate refinement of models specified in UML/OCL. The paper describes the development of an access control system in a series of steps, each step is a more detailed (refined) model of the previous step. The development is similar to how one would develop a system in B or in Event-B, but in contrast to these systems, the paper does not offer a concrete set of rules for checking that the refinement is valid. More recently [18] they suggest to verify refinement by translating the UML class diagrams and OCL constraints into a B model.

Ducournau et al. [19] describe the refinement of individual classes (but not of associations), in contrast to our approach,

which is refinement of entire models. Its definition of refinement is informal.

Shen et al. [20] describe a UML metamodel for checking refinement between class diagrams. Refinement is defined between individual classes and associations. Associations can only be refined to concrete chains. They focus on a metamodel framework for enforcing refinement rules. Defining the rules is left to the developers. Later work [21] discusses a technique for checking that a more concrete class diagram is consistent with an abstract one. They compute an abstraction of the concrete diagram and compare it with the abstract diagram. If the relationships in the two models differ, the models are deemed inconsistent. To generate the abstraction they compute the transitive closure of the relationships in the concrete diagram, and keep only the relationships between classes that appear in the abstract diagram.

There are several important differences between this work and ours. First, our notion of refinement allows the concrete model to restrict the abstract model, whereas [21] demands exact equivalence. Second, we support more ways in which concrete entities may represent an abstract entity. In particular, in diagram refinement several associations in a concrete diagram may represent the same abstract association. The concrete model of our first example (that has *mother* and *father* associations) would not be considered consistent with its abstract version according to [21]. Third, diagram refinement extends to other kinds of UML diagrams, whereas [21] applies only to class diagrams. Finally, because we use an explicit mapping between the concrete and the abstract models, diagram refinement is much easier to check. Indeed, it is feasible to check diagram refinement manually, whereas the rules defined in [21] must be checked by a machine for all but the smallest models.

Pons et-al. [22] discuss the refinement of classes and associations, including the case where several concrete associations refine one abstract association (heterogeneous refinement). The paper illustrates refinement of individual constructs (classes, associations), but has no rules for determining valid refinement in the general case. In addition, it ignores multiplicities.

Cruz et-al. [23] discuss refinement of use cases and use case diagrams. Like our work, they argue that the use cases of a large system should be described by separate diagrams at different levels of abstractions, with a refinement relation that links the concrete models to the abstract models. However, they do not consider *include* and *extend* relationships between use cases, and their discussion is informal.

Model subtyping [24] facilitates the reuse of model transformations. Given a model transformation of a model A, we may safely apply the transformation to any model that is a subtype of A. Both diagram refinement and model subtyping are partial orders between models, but they are different because they have different goals. For example, the concrete diagram in Fig. 1 is not a subtype of the abstract diagram because any model transformation that expects to access the abstract *parents* relation will break on the concrete diagram which does not contain this association.

Model slicing [25] is an abstraction technique that facilitates the verification of model properties by extracting from a model only those parts that are relevant to the property being checked. Model slicing is substantially different from diagram refinement, first because it is applied to behavioral models, and second because it requires a specific goal (e.g. a particular transition in a state diagram) and throws away anything that is not relevant to that goal. In contrast, diagram refinement is a relationship between a detailed model and its summary. Nothing disappears, but some things will appear in less details.

VIII. Discussion

We have described a set of rules for relating concrete diagrams to abstract diagrams. The rules can be understood solely in terms of their informal meaning, which makes them suitable for developers that are not proficient with formal mathematical foundations. The mathematics that underlies the rules is used to ensure that they are consistent, simple, and sound. We insist, e.g., that every class in the abstract diagram is mapped to exactly one class in the concrete diagram (rule 1). One may ask why not relax the rule and allow many concrete classes to be mapped to the same abstract class? We have started our investigation with exactly this idea, but the mathematical analysis has revealed many difficulties and complications with this direction. Upon further reflection we have noticed that whenever several concrete concepts $c_1, \ldots, c_n$ represent the same abstract concept a, we can always add a to the concrete class diagram and have each c_i inherit from a. Therefore we do not lose anything by insisting on the uniqueness rule, and in return greatly simplify our framework.

The mathematical framework that we have developed has two parts. The first part is a generic notion of refinement defined in terms of graph homomorphisms—mathematical transformations between graphs that preserve their structure. This notion of refinement ensures that whenever a graph represents a relational model, any valid refinement of the graph yields a more detailed relational model that is consistent with the original model. The second part, which is unique to every diagram type, is the relational semantics that we define for each diagram type. By separating the mathematical framework into two parts, we make it possible to support additional diagram types with a single formal notion of refinement. As long as we can develop a relational semantics that represents the relationships in the diagram, we can use diagram refinement to help us develop such diagrams incrementally and consistently.

In the future we plan to implement a prototype tool to automatically check for diagram refinement, and to support additional diagram types. In addition, we plan to extend the theory to deal with maintaining consistency between several concrete diagrams that refine the same abstract diagram.

Acknowledgment

We would like to thank Mr. Tony Linde for letting us use his AstroGrid conceptual domain diagram in this paper. This work has been partially supported by GIF (grant No. 1131-9.6/2011).

REFERENCES

[1] B. Bruegge and A. H. Dutoit, *Object-Oriented Software Engineering Using UML, Patterns, and Java*, 3rd ed. Prentice Hall, 2009.

[2] H. Podeswa, *UML for the IT Business Analyst: A Practical Guide to Object-Oriented Requirements Gathering*. Thomson Course Technology Press, 2005.

[3] J. Rumbaugh, I. Jacobson, and G. Booch, *Unified Modeling Language Reference Manual*. Pearson Higher Education, 2004.

[4] Z. Liu, X. Li, J. Liu, and J. He, "Integrating and refining UML models," in *Workshop on Consistency Problems in UML-Based Software Development: Understanding and Usage of Dependency Relationships*, 2004, pp. 23–40.

[5] J. Simmonds, R. V. D. Straeten, V. Jonckers, and T. Mens, "Maintaining consistency between UML models using description logic," *Série L'objet*, vol. 10, no. 2-3, pp. 231–244, 2004.

[6] C. Snook and M. Butler, "UML-B: Formal modeling and design aided by UML," *ACM Transactions on Software Engineering and Methodology*, vol. 15, no. 1, pp. 92–122, 2006.

[7] Z. Diskin and T. S. E. Maibaum, "Category theory and model-driven engineering: From formal semantics to design patterns and beyond," in *Workshop on Applied and Computational Category Theory (ACCAT)*, 2012, pp. 1–21.

[8] K. Lano and J. Bicarregui, "UML refinement and abstraction transformations," in *Workshop on Rigorous Object Orientated Methods (ROOM2)*, 1998.

[9] C. Pons, "Heuristics on the definition of UML refinement patterns," in *Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM)*, ser. LNCS, J. Wiedermann, G. Tel, J. Pokorný, M. Bieliková, and J. Stuller, Eds., vol. 3831. Springer, 2006, pp. 461–470.

[10] J. Woodcock and J. Davies, *Using Z: Specification, Refinement, and Proof*. Prentice-Hall, 1996.

[11] J. Abrial, *The B-book: Assigning programs to meanings*. Cambridge Press, 2005.

[12] M. Petre, "Uml in practice," in *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press, 2013, pp. 722–731.

[13] T. Linde, "AstroGrid," `http://tinyurl.com/hrz6va8`, 2009, accessed: August 2016.

[14] B. Hnatkowska, Z. Huzar, and L. Tuzinkiewicz, "On understanding of refinement relationship," in *Workshop on Consistency Problems in UML-based Software Development: Understanding and Usage of Dependency Relationships*, 2004, pp. 7–18.

[15] A. S. Evans, "Reasoning with UML class diagrams," in *Workshop on Industrial Strength Formal Specification Techniques (WIFT)*. IEEE, 1998, pp. 102–113.

[16] K. Bergner, A. Rausch, M. Sihling, and A. Vilbig, "Structuring and refinement of class diagrams," in *Hawaii International Conference on System Sciences (HICSS)*. IEEE, 1999.

[17] B. Ben Ammar, M. T. Bhiri, and J. Souquières, "Towards an incremental development of UML specifications," 2007, Rapport.

[18] ——, "Incremental development of UML specifications using operation refinements," *Innovations in Systems and Software Engineering (ISSE)*, vol. 4, no. 3, pp. 259–266, 2008.

[19] R. Ducournau, F. Morandat, and J. Privat, "Modules and class refinement: a metamodeling approach to object-oriented languages," Université Montpellier 2, Tech. Rep. LIRMM-07021, 2007.

[20] W. Shen and W. L. Low, "Using the metamodel mechanism to support class refinement," in *International Conference on Engineering of Complex Computer Systems (ICECCS)*. IEEE, 2005, pp. 421–430.

[21] W. Shen, K. Wang, and A. Egyed, "An efficient and scalable approach to correct class model refinement," *Trans. Software Eng.*, vol. 35, no. 4, pp. 515–533, 2009.

[22] C. Pons, G. Pérez, R. S. Giandini, and R. Kutsche, "Understanding refinement and specialization in the UML," in *Workshop on MAnaging SPEcialization/Generalization Hierarchies (MASPEGHI)*, 2003.

[23] E. F. Cruz, R. J. Machado, and M. Y. Santos, "On the decomposition of use cases for the refinement of software requirements." in *International Conference on Computational Science and Its Applications (ICCSA)*, B. O. Apduhan, A. M. A. C. Rocha, S. Misra, D. Taniar, O. Gervasi, and B. Murgante, Eds., 2014, pp. 237–240.

[24] J. Steel and J.-M. Jézéquel, "On model typing," *Software & Systems Modeling*, vol. 6, no. 4, pp. 401–413, 2007.

[25] K. Androutsopoulos, D. Clark, M. Harman, J. Krinke, and L. Tratt, "State-based model slicing: A survey," *ACM Computing Surveys*, vol. 45, no. 4, pp. 53:1–53:36, Aug. 2013.