Consistent Adaptation and Evolution of Class Diagrams during Refinement

Alexander Egyed

Teknowledge Corporation, 4640 Admiralty Way, Suite 1010
Marina Del Rey, CA 90034, USA
aegyed@ieee.org

Abstract. Software models are key in separating and solving independent development concerns. However, there is still a gap on how to transition design information among these separate, but related models during development and maintenance. This paper addresses the problem on how to maintain the consistency of UML class diagrams during various levels of refinement. We present a new approach to automated consistency checking called *ViewIntegra*. Our approach separates consistency checking into transformation and comparison. It uses transformation to translate model elements to simplify their subsequent comparison. Transformation-based consistency checking, in the manner we use it, is new since we use transformation to bridge the gap between software models. No intermediate models or model checkers are required; developers need only be familiar with the models they design with and none other. The separation of transformation and comparison makes our approach to consistency checking more transparent. It also makes our approach useful for both propagating design changes among models and validating consistency. This gives developers added flexibility in deciding when to re-generate a model from scratch or when to resolve its inconsistencies. Although this paper emphasizes the adaptation and evaluation of class diagrams, we found our technique to be equally useful on other models. Our approach is tool supported.

1 Introduction

In the past decades, numerous software models were created to support software development at large. Models usually break up software development into smaller, more comprehensible pieces utilizing a divide and conquer strategy. The major drawback of models is that development concerns cannot truly be investigated all by themselves since they depend on one another. If a set of issues about a system is investigated, each through its own models, then the validity of a solution derived from those models requires that commonalities (redundancies) between them are recognized and maintained in a consistent fashion. Maintaining consistency between models is a nontrivial problem. It is expensive and labor intensive despite the vast number of past and existing research contributions.

M. Wermelinger and T. Margaria-Steffen (Eds.): FASE 2004, LNCS 2984, pp. 37–53, 2004.

In this work, we introduce a transformation-based approach to consistency checking called *ViewIntegra*. This paper describes our approach to automated consistency checking and shows how to translate models to simplify their comparison. The effect is that redundant information from one model is re-interpreted in the context and language of another model followed by a simple, one-to-one comparison to detect differences. We limit the discussion in this paper to the abstraction and refinement of UML-like class diagrams [24]. We believe our approach to be equally applicable to other kinds of models.

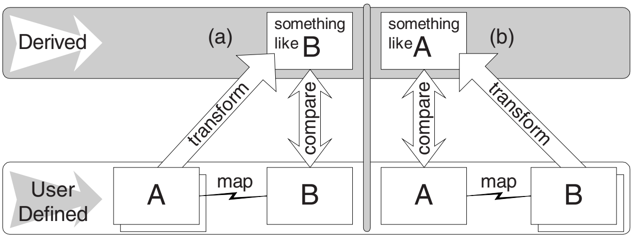


Fig. 1. View transformation and mapping to complement view comparison

Figure 1 depicts the principle of transformation-based consistency checking. In order to compare the two (user-defined[1]) models A and B (e.g., high-level model and low-level model), we transform one of them into 'something like the other' so that the one becomes easier comparable to the other. For example, our approach transforms the low-level class diagram into a form that makes the results directly comparable with the high-level diagram. As part of our collaboration with Rational Corporation we created such an transformation technique [9] and, in this paper, we will demonstrate how its results can be used for direct, one-to-one comparison to detect inconsistencies.

Our approach separates the propagation of design information (transformation) from the comparing of design information (consistency checking). It follows that transformation may be used independently from comparison for change propagation. For example, the above mentioned transformation technique can be used during reverse engineering to generate a high-level class diagram from a lower-level one; or the transformation technique can be used during consistency checking to suggest its transformation results as an option for resolving inconsistencies.

Traceability among model elements is needed to guide transformation and comparison. We found that it is typically very hard to generate traceability information in detail although developers are capable of approximating it [6]. This would be a problem for any consistency checking approach but our approach can alleviate this problem significantly. This paper will thus also demonstrate how our approach behaves with a partial lack of traceability information.

[1] User-defined views are diagrams that are created by humans (e.g., Fig. 2.). Derived views (interpretations) are diagrams that are automatically generated via *Transformation*.

Because our approach separates transformation from comparison during consistency checking, it also benefits from reuse of previously transformed, unchanged design information. This greatly improves performance because subsequent comparisons require partial re-transformations only. Another benefit is that consistency rules are very generic and simple since they have to compare alike model elements only.

We evaluated our transformation-based approach to consistency checking on various types of heterogonous models like class diagrams, state chart diagrams, sequence diagrams [8], and the C2SADEL architecture description language [10]. Furthermore, we validated the usefulness of our approach (in terms of its scope), its correctness (e.g., true errors, false errors), and scalability via a series of experiments using third-party models [1,2] and in-house developed models. Our tool support, called UML/Analyzer, fully implements transformation-based consistency checking for class and C2SADEL diagrams. The other diagrams are partially supported only.

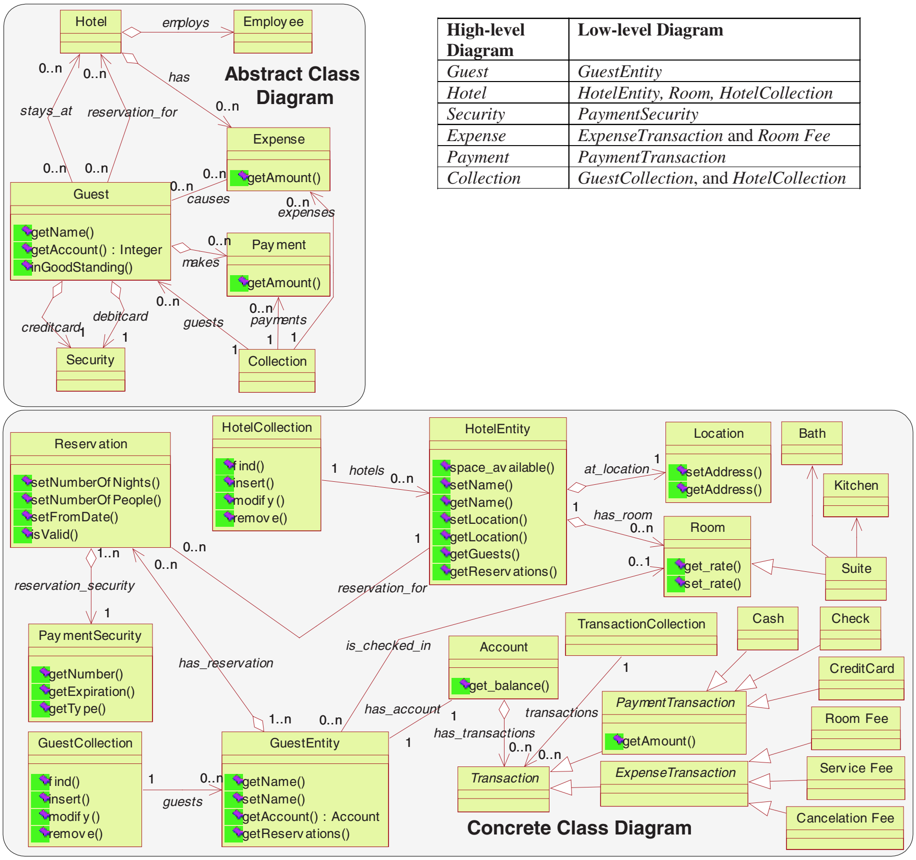
The remainder of this paper is organized as follows: Section 2 introduces an example and discusses abstraction and refinement problems in context of two class diagrams depicted there. Section 3 will highlight consistency checking without transformation and discusses in what cases it is effective and where it fails. Section 4 introduces a transformation technique for abstracting class diagrams and discusses how it improves the scope of detectable inconsistencies. Section 4 discusses how our transformation and consistency checking methods are also able to interpret incomplete and ambiguous model information. Section 5 discusses issues like scope, accuracy, and scalability in more detail and Section 6 summarizes the relevance of our work with respect to related work.

2 Example

Figure 2 depicts two class diagrams of a *Hotel Management System* (HMS) at two levels of abstraction. The top diagram is the higher-level class diagram with classes like *Guest* and *Hotel*, and relationships like "a guest may either stay at a hotel or may have reservations for it." This diagram further states that a *Hotel* may have *Employees* and that there are *Expense* and *Payment* transactions associated with guests (and hotels). It is also indicated that a *Guest* requires a *Security* deposit. The diagram uses three types of relationships to indicate uni-directional, bi-directional, and part-of dependencies (see UML definition [24]). For instance, the relationship with the diamond head indicates aggregation (part-of) implying that, say, *Security* is a part of *Guest*. Additionally, the diagram lists a few methods that are associated with classes. For instance, the class *Expense* has one method called *getAmount()*.

The bottom part of Figure 2 depicts a refinement of the left side. Basic entities like *Guest* or *Expense* are still present although named slightly differently[2] and additional classes were introduced. For instance, the lower-level diagram makes use of new classes like *Reservation* or *Check* to refine or extend the higher-level diagram. The

[2] It must be noted that we use a disjoint set of class names in order to avoid naming confusions throughout this paper. Duplicate names are allowed as part of separate name spaces.



High-level Diagram	**Low-level Diagram**
Guest	*GuestEntity*
Hotel	*HotelEntity, Room, HotelCollection*
Security	*PaymentSecurity*
Expense	*ExpenseTransaction* and *Room Fee*
Payment	*PaymentTransaction*
Collection	*GuestCollection*, and *HotelCollection*

Fig. 2. High-level and low-level class diagrams of HMS and mapping table

refined class diagram uses the same types of relationships as the high-level one plus generalization relationships (triangle head) to indicate inheritance. The low-level diagram also describes methods associated with classes more extensively.

Both diagrams in Figure 2 separately describe the structure of the HMS system. Obviously, there must be some commonality between them (i.e., redundancy). For instance, the high-level class *Guest* is equivalent to the low-level class *GuestEntity* and the high-level relationship "*has*" (between *Guest* and *Expense*) is equivalent to the combined relationships with the classes *Transaction* and *Account* between them. The table in Figure 2 describes several cases of one-to-many mappings such as Hotel maps to *HotelEntity*, *Room*, *HotelCollection*, one case of a many-to-many mapping (there are several choices of how *reservation_for* and *stays_at* map to the low-level diagram), one case of a many-to-one mapping (*HotelCollection* is assigned to *Hotel* and *Collection*), and many cases of no mappings altogether (e.g., *Employee* in the high-level diagram or *Account* in the low-level diagram).

Knowledge on how model elements in separate diagrams relate to one another is commonly referred to as traceability (or mapping) [13]. Traceability is usually generated manually [16] (either by using common names or maintaining trace tables as depicted in Figure 2) but there exists some automation [6]. Like other consistency checking approaches, our approach requires some traceability knowledge but a discussion on how to derive it is out of the scope of this paper.

3 Simple Consistency Checking

Current consistency checking approaches detect inconsistencies by transforming models into some third-party language followed by constraint-based reasoning (often model checking) in context of that language. For instance, consistency checking approaches like JViews (MViews) [14], ViewPoints [15] or VisualSpecs [4] read diagrams, translate them into common (and usually formal) representation schemes, and validate inconsistency rules against them. These approaches have shown powerful results; they are precise and computationally efficient. But there are also unresolved side effects that one could well argue to be outside the scope of consistency checking but that are related and significant:

(1) Lack of Change Propagation: Existing approaches solve the problem of detecting inconsistencies very well but they lack support for the subsequent, necessary adaptation of models once inconsistencies are found.

(2) Lack of Traceability: Existing approaches require complete traceability to guide consistency checking. Generating traceability is a hard, error-prone activity with a potentially little life span.

4 Transformation-Based Consistency Checking

We describe our approach to automated consistency checking next and also discuss how it is able to handle above side effects. Our approach, called *ViewIntegra* [8], exploits the redundancy between models: for instance, the high-level diagram in Figure 2 contains information about the HMS system that is also captured in the low-level diagram. This redundant information can be seen as a constraint. Our approach uses transformation to translate redundant model information to simplify their comparison. The effect is that redundant information from one model is re-interpreted in the context and language of another model followed by a simple, one-to-one comparison to detect differences (effectively enforcing the constraint).

Abstraction Implementing Transformation

In the course of evaluating nine types of software models [8] (class, object, sequence, and state chart diagrams, their abstractions and C2SADEL) we identified the need for four transformation types called *Abstraction, Generalization*, *Structuralization*, and

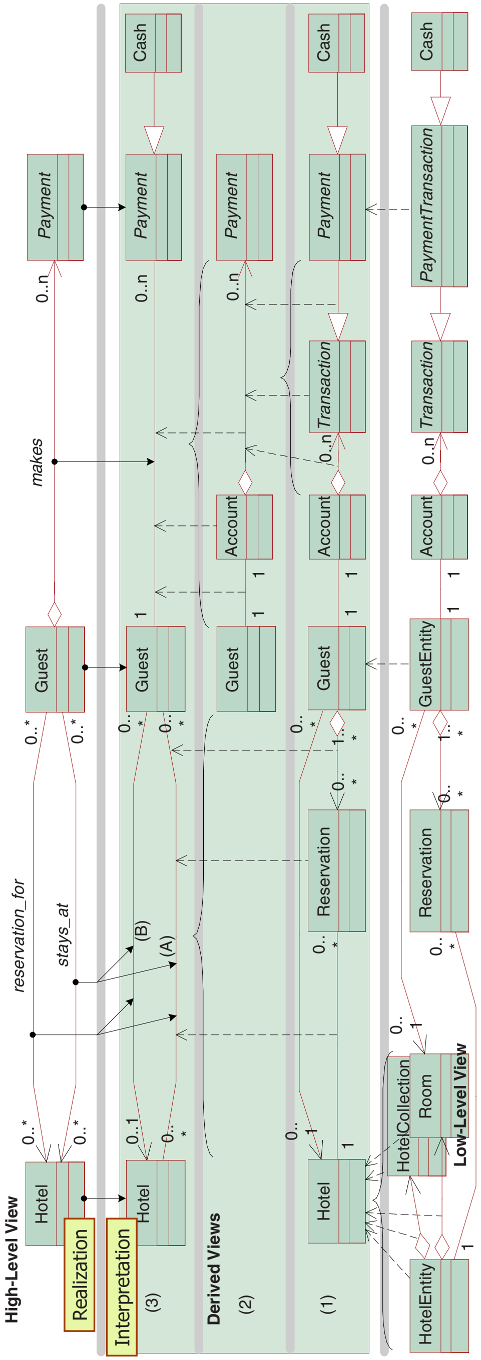


Fig. 3. Abstraction applied on part of HMS showing high-level, low-level, and derived modeling information

Translation. This paper focuses on inconsistencies during refinement and thus only needs *Abstraction*. See [8] for a discussion on the other types.

Abstraction deals with the simplification of information by removing details not necessary on a higher level. In [7], we identified two types of abstractions called *compositional abstraction* and *relational abstraction*. Compositional abstraction is probably the more intuitive abstraction type since it closely resembles hierarchical decomposition of systems. For instance, in UML, a tree-like hierarchy of classes can be built using a feature of classes that allows them to contain other classes. Thus, a class can be subdivided into other classes. In relational abstraction it is the relations (arrows) and not the classes that serve as vehicles for abstraction. Relations (with classes) can be collapsed into more abstract relations. Relational abstraction is needed since it is frequently not possible to maintain a strict tree-hierarchy of classes. Our abstraction technique has been published previously; we will provide a brief summary here only. For a more detailed discussion, please refer to [7,9].

In order to abstract the low-level diagram in Figure 2, we have to apply both abstraction types. Figure 3 shows a partial view of Figure 2 depicting, in the top layer, the high-level classes *Hotel*, *Guest*, and *Payment* and, in the bottom layer, their low-level counterparts *HotelEntity*, *HotelCollection*, *Room*, *GuestEntity*, and *PaymentTransaction*. The bottom layer also depicts relationships among the low-level classes.

Table 1. Excerpt of abstraction rules for classes [9]

1) Class x Association x Class x AggregationRight x Class equals Association 100
2) Class x AggregationLeft x Class x AssociationLeft x Class equals AssociationLeft 100
3) Class x Association x Class x AggregationLeft x Class equals Association 90
4) Class x AggregationLeft x Class x GeneralizationLeft x Class equals AggregationLeft 100
5) Class x GeneralizationLeft x Class x GeneralizationLeft x Class equals GeneralizationLeft 100
6) Class x DependencyRight x Class x AggregationRight x Class equals DependencyRight 100
7) Class x AssociationRight x Class x GeneralizationRight x Class equals AssociationRight 70
8) Class x Aggregation x Class equals Class 100

The first abstraction step is to use compositional abstraction to group low-level classes that belong to single high-level classes. For instance, the low-level classes *HotelEntity*, *HotelCollection*, and *Room* are all part of the same high-level class *Hotel* (recall traceability in Figure 2). The grouped class, called *Hotel*, is depicted in the first (1) derived view in Figure 3. Besides grouping the three low-level classes, the abstraction method also replicated the inter-dependencies of those three classes for the derived, high-level class. It can be seen that the derived class *Hotel* now has relationships to *Reservation* and *Guest* that were taken from *HotelEntity* and *Room* respectively. Also note that the single low-level classes *GuestEntity* and *PaymentTransaction* were grouped into the more high-level, derived classes *Guest* and *Payment*. They also took all inter-relationships from their low-level counterparts. Compositional abstraction simplified the low-level class diagram somewhat but it is still not possible to compare it directly to the high-level diagram. To simplify comparison, helper classes such as *Reservation*, *Account*, and *Transaction* need to be eliminated since they obstruct our understanding on the high-level relationships between classes such as *Hotel* and *Guest*. The problem is that those classes were not assigned to any high-level classes and thus could not be eliminated via compositional abstraction.

The second abstraction step is to group low-level relationships into single high-level relationships. For instance, the low-level relationship path going from *Guest* via *Reservation* to *Hotel* in Figure 3 (bottom) may have some abstract meaning. This meaning can be approximated through simpler, higher-level model elements. In particular, this example shows an aggregation relationship between the classes *Reservation* and *Guest* (diamond head) indicating that *Reservation* is a part of *Guest*. The example also shows a uni-directional association relationship from *Hotel* to *Reservation* indicating that *Reservation* can access methods and attributes of *Hotel* but not vice versa. What the diagram does not depict is the (more high-level) relationship between *Guest* and *Hotel*. Semantically, the fact that *Reservation* is part of a *Guest* implies that the class *Reservation* is conceptually *within* the class *Guest*. Therefore, if *Reservation* can access *Hotel*, *Guest* is also able to access *Hotel*. It follows that *Guest* relates to *Hotel* in the same manner as *Reservation* relates to *Hotel* making it possible for us to replace *Reservation* and its relationships with a single *association* originating in *Guest* and terminating in *Hotel* (third derived view in Figure 3).

In the course of inspecting numerous UML-type class diagrams, we identified over 120 class abstraction rules [9]. Table 1 shows a small sample of these abstraction rules as needed in this paper. Abstraction rules have a *given* part (left of "equals") and an *implies* part (right of "equals"). Rules 1 and 8 correspond to the two rules we discussed so far. Since the directionality of relationships is very important, the rules in Table 1 use the convention of extending the relationship type name with “Right” or “Left” to indicate the directions of their arrowheads. Furthermore, the number at the end of the rule indicates its reliability. Since rules are based on semantic interpretations, those rules may not always be valid. We use reliability numbers as a form of priority setting to distinguish more reliable rules from less reliable ones. Priorities are applied when deciding what rules to use when. We will later discover that those reliability numbers are also very helpful in determining false errors. It must be noted that rules may be applied in varying orders and they may also be applied recursively. Through recursion is it possible to eliminate multiple helper classes as in the case of the path between *PaymentTransaction* and *GuestEntity* (see second (2) and third (3) derived views in Figure 3).

Compositional and relational abstraction must be applied to the entire low-level diagram (Figure 2). A part of the resulting abstraction is depicted in the second row in Figure 3 (third (3) view). We refer to this result as the *interpretation* of the low-level diagram. This interpretation must now be compared to the high-level diagram.

Comparison

The abstraction technique presented above satisfies our criteria of a good transformation method because it transforms a given low-level class diagram into ‘something like’ the high-level class diagram. Obviously, consistency checking is greatly simplified because a straightforward, one-to-one comparison will detect inconsistencies. This section introduces consistency rules for comparison. The beginning of Section 4 listed the lack of traceability as a major challenge during consistency checking. This section shows how to identify inconsistencies, and in doing so, how to handle missing traceability. In the following, we will first describe the basics of comparison and how to handle traceability under normal conditions. Thereafter, we will discuss ambiguous reasoning to handle missing traceability. In the following we will refer to the *interpretation* as the abstracted (transformed) low-level class diagram and to the *realization* as the existing high-level class diagram. The goal of consistency checking is to compare the realization with the interpretation.

Before transformation, we knew about (some) traceability between the high-level diagram (realization) and the low-level diagram but no traceability is known between the realization and the interpretation. This problem can be fixed easily. Any transformation technique should be able to maintain traceability between the transformation result (interpretation) and the input data (low-level diagram). This is easy because transformation knows what low-level model elements contribute to the interpretation. Through transitive reasoning, we then derive traceability between the realization and the interpretation. For example, we know that the derived *Hotel* is the result of grouping {*HotelEntity*, *Room*, *HotelCollection*} (see dashed arrows in Figure 3) and we know that this group traces to the high-level *Hotel* (mapping table in Figure 2). Thus, there is a transitive trace dependency between the class *Hotel* in the realization and the *Hotel* in the interpretation. Arrows with circular arrowheads in Figure 3 show these transitive trace dependencies between the realization and interpretation.

Ideally, there should be one-to-one traces between realization and interpretation elements only. Unfortunately, partial knowledge about trace dependencies may result in one-to-many dependencies or even many-to-many dependencies (e.g., realization relation *reservation_for* traces to two relationships in the interpretation). This is represented with a fork-shaped trace arrow in Figure 3.

In the following we present a small sample of consistency rules relevant in this paper. Consistency rules have two parts; a qualifier to delimit the model elements it applies to and a condition that must be valid for the consistency to be true[3]:

1. Type of low-level relation is different from abstraction:

```
∀ r ∊ relations, interpretation(r)≠null ⇒ type(interpretation(r))=type(r)
```

Rule 1 states that for a relation to be consistent it must have the same type as its corresponding interpretation. Its qualifier (before "⇒") defines that this rule applies to relations only that have a known interpretation. The traceability arrow in Figure 3 defines such known interpretations (or in reverse known realizations). In Figure 3, we have six interpretation traces; three of which are originating from relationships (circular ends attached to lines): the realization relations "stays_at" and "reservation_for" satisfy above condition[4], however, the realization relation "makes" does not. The latter case denotes an inconsistency because "makes" is of type "aggregation" and its interpretation is of type "association." If we now follow the abstraction traces backward (dashed arrows, that were generated during abstraction), it becomes possible to identify the classes *Account* and *Transaction* as well as their relationships to *GuestEntity* and *PaymentTransaction* a having contributed to the inconsistent interpretation.

2. Low-level relation has no corresponding abstraction:

```
∀ r ∊ relations, abstractions(r)->size=0 ∧ realizations(r)=null ⇒
¬[∃ c ∊ classes(r), realizations(c)≠null]
```

Rule 2 states that all (low-level) relations must trace to at least one high-level model element. To validate this case, the qualifier states that it applies (1) to relations that do not have any *abstractions* (dashed arrows) and (2) to relations that do not have *realizations*. Figure 3 has many relations (derived and user-defined ones). Checking for relations that do not have abstractions ensures that only the most high-level, abstracted relations are considered; ignoring low-level relations such as the aggregation from *Transaction* to *Account*. The rule thus defines that consistency is ensured if *none* of the classes attached to the relation have realizations themselves. The generalization from *Cash* to *Payment* in Figure 3 violates this rule. This generalization neither has an abstraction nor a realization trace but its attached class *Payment*

[3] Some qualifier conditions were omitted for brevity (e.g., checking for transformation type) since they are not needed here.

[4] For now treat the one-to-many traces as two separate one-to-one traces. We will discuss later how to deal with it properly.

has a realization trace[5]. This example implies that the high-level diagram does not represent the relationship to *Cash* or that traceability about it is unknown.

3. Destination direction/navigability of relation does not match abstract relation:

```
∀ r ∈ relations, interpretation(r)≠null ∧
type(interpretation(r))=type(r) ⇒ [size(r->destClass ÷
realization(interpretation(r)->destClass))=0]
```

Rule 3 defines that for two relations to be consistent they ought to be pointing in the same directions (same destination classes). This rule applies to relations that have interpretations and to relations that have the same type. It defines that the realization "r" must have the same destination classes as the realizations of the interpretation's destination classes. A destination class here is a class at the end of a relation's arrowhead (e.g., *Hotel* for *reservation_for*). This rule applies to the two relations *reservation_for* and *stays_at* only (the relation *makes* is ruled out since the qualifier requires relations to be of the same type).

Ambiguous Reasoning

Comparison is not sufficient to establish consistency correctly. Rule 3 applied to the realization relations *reservation_for* and *stays_at* results in consistency being true for both cases. This is misleading because the traceability is ambiguous in that the two high-level (realization) relations point to the same two interpretations (labeled (A) and (B) in Figure 3). The problem is caused by the lack of traceability. Our approach addresses this problem by hypothesizing that at least one of the potentially many choices ought to be consistent. Thus, comparison attempts to find one interpretation for *reservation_for* and one for *stays_at* that is consistent. If no interpretation is consistent then there is a clear inconsistency in the model. If exactly one interpretation is consistent then this interpretation must be the missing trace (otherwise there would be an inconsistency). Finally, if more than one interpretation is consistent then the situation remains ambiguous (although potentially less ambiguous since inconsistent interpretations can still be eliminated as choices). Should more than one consistency rule apply to a model element then all of them need to be satisfied. Each constraint may thus exclude any inconsistent interpretation it encounters.

For instance, in case of the relation *reservation_for*, our approach compares it with both interpretations (A) and (B). It finds rule 3 to be inconsistent if the relation *reservation_for* is compared to interpretation (A); and it finds the rule to be consistent if it is compared to (B). Our approach thus eliminates the trace to interoperation (A) as being incorrect (obviously it leads to inconsistency which cannot be correct). What remains is an ideal, one-to-one mapping. Our approach then does the same for the realization *stays_at* with the result this it is also inconsistent with interpretation (A) and consistent with interpretation (B). Again, the trace to the inconsistent interpretation is removed.

[5] It is outside the scope of this paper to discuss the workings of our reduced redundancy model which treats derivatives like *Payment* together with *PaymentTransaction* as "one element."

Ambiguous reasoning must ensure that every realization has exactly one interpretation it does not share with another realization. For example, in the previous two evaluations we found exactly one interpretation for both realizations *reservation_for* and *stays_at*; however, in both cases it is the same interpretation. This violates one-to-one comparison. Recall that transformation ensures that model elements become directly comparable. Every realization must have exactly one interpretation. To resolve this problem we have to identify the conflicting use of the same interpretations: this is analogous to the resource allocation problem which handles the problem on how to uniquely allocate a resource (resource = interpretation). The maximum flow algorithm [5] (Ford-Fulkerson [12]) solves the resource allocation problem efficiently. The algorithm can be applied to undirected graphs (true in our case since traceability links are undirected) where the algorithm guarantees a maximum matching of edges (traces) without the same vertex (model elements) being used twice. In short, the maximum-bi-partite-matching problem can be used to avoid the duplicate use of interpretations. In the previous example, the algorithm is not able to find a solution that satisfies both realizations. It thus detects an inconsistency.

It must be noted at this point that our ambiguity resolution mechanism has an element of randomness in that the outcome may vary if the order, in which model elements are validated, differs. As such, the maximum bi-partite algorithm will use interpretation (B) for either *stays_at* or *reservation_for* and report a resource conflict (~inconsistency) for the other.

In summary, validating the consistency among model elements potentially encounters three situations as depicted in Figure 4. Situation (a) is the most simplistic one where there is a one-to-one mapping between interpretation (I) and realization (R). The example discussed in Rule 1 above showed such a case. Situation b) corresponds to the example we discussed with Rule 2 where we encountered a low-level interpretation (low-level relation) that had no abstraction. The reverse is also possible where there is a high-level realization that has no refinement. While discussing Rule 3 with its ambiguity example, we encountered situation c) where one realization had two or more interpretations (one-to-many mapping). This scenario required the validation of consistency on all interpretations (OR condition). Traces to inconsistent interpretations were removed and the maximum-partite algorithm was used to find a configuration that resolved all remaining ambiguities randomly.

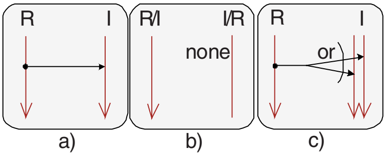


Fig. 4. Basic Comparison Rules for Ambiguity

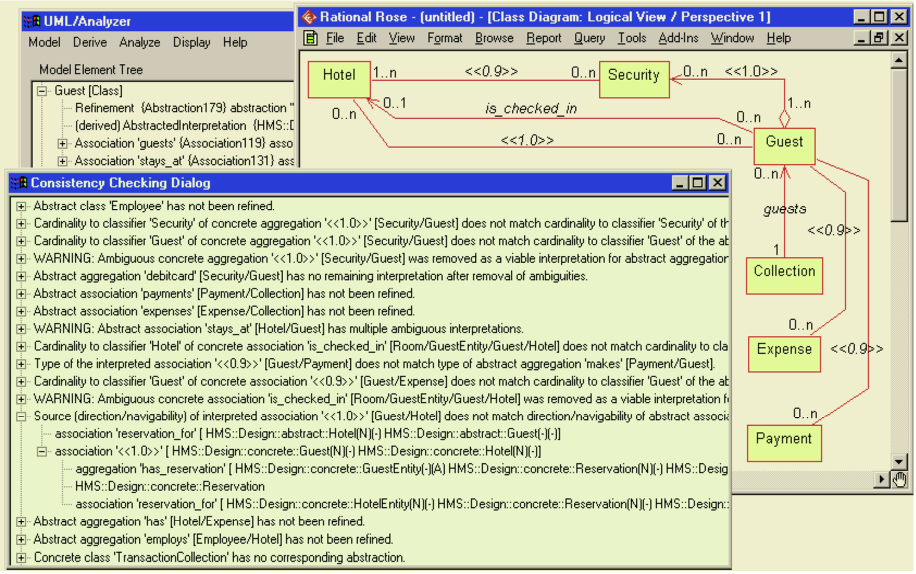
5 Discussion

Scope

In addition to the (in)consistency rules presented in this paper, we identified almost 20 more that apply to refinement [8]. Figure 5 (bottom left) shows an excerpt of inconsistencies between the diagrams in Figure 2 as generated by our tool *UML/Analyzer*. Our tool is also integrated with Rational Rose® which is used as a graphical front-end. The right side depicts the complete derived abstraction of the low-level diagram (Figure 2). Partially hidden in the upper left corner of Figure 5 is the UML/Analyzer main window, depicting the repository view of our example. Besides inconsistency messages, our tool also gives extensive feedback about the model elements involved. For instance, in Figure 5 one inconsistency is displayed in more detail, revealing three low-level model elements (e.g., *Reservation*) as the potential cause of the inconsistency. We also identified around 40 additional inconsistency types between other types of UML diagrams [8] (sequence and state chart diagrams) and the non-UML language C2SADEL [10].

Accuracy (True Inconsistencies/False Inconsistencies)

An important factor on how to estimate the accuracy of any consistency checking approach is in measuring how often it provides erroneous feedback (e.g., report of inconsistencies were there are none or missing inconsistencies). As any automated inconsistency detection approach, our approach may not produce correct results at all times. However, our approach provides means of evaluating the level of "trust" one may have in its feedback. For instance, in Table 1 we presented abstraction rules and



Model Element Tree

```
⊟ Guest [Class]
    Refinement  {Abstraction179} abstraction "
    (derived) AbstractedInterpretation  {HMS::D
  ⊞ Association 'guests' {Association119} asso
  ⊞ Association 'stays_at' {Association131} ass
```

```
⊞ Abstract class 'Employee' has not been refined.
⊞ Cardinality to classifier 'Security' of concrete aggregation '<<1.0>>' [Security/Guest] does not match cardinality to classifier 'Security' of th
⊞ Cardinality to classifier 'Guest' of concrete aggregation '<<1.0>>' [Security/Guest] does not match cardinality to classifier 'Guest' of the ab
⊞ WARNING: Ambiguous concrete aggregation '<<1.0>>' [Security/Guest] was removed as a viable interpretation for abstract aggregatior
⊞ Abstract aggregation 'debitcard' [Security/Guest] has no remaining interpretation after removal of ambiguities.
⊞ Abstract association 'payments' [Payment/Collection] has not been refined.
⊞ Abstract association 'expenses' [Expense/Collection] has not been refined.
⊞ WARNING: Abstract association 'stays_at' [Hotel/Guest] has multiple ambiguous interpretations.
⊞ Cardinality to classifier 'Hotel' of concrete association 'is_checked_in' [Room/GuestEntity/Guest/Hotel] does not match cardinality to cla
⊞ Type of the interpreted association '<<0.9>>' [Guest/Payment] does not match type of abstract aggregation 'makes' [Payment/Guest].
⊞ Cardinality to classifier 'Guest' of concrete association '<<0.9>>' [Guest/Expense] does not match cardinality to classifier 'Guest' of the ab
⊞ WARNING: Ambiguous concrete association 'is_checked_in' [Room/GuestEntity/Guest/Hotel] was removed as a viable interpretation fi
⊟ Source (direction/navigability) of interpreted association '<<1.0>>' [Guest/Hotel] does not match direction/navigability of abstract associa
    association 'reservation_for' [ HMS::Design::abstract::Hotel(N)(-) HMS::Design::abstract::Guest(-)(-)]
  ⊟ association '<<1.0>>' [ HMS::Design::concrete::Guest(N)(-) HMS::Design::concrete::Hotel(N)(-)]
      aggregation 'has_reservation' [ HMS::Design::concrete::GuestEntity(-)(A) HMS::Design::concrete::Reservation(N)(-) HMS::Desig
      HMS::Design::concrete::Reservation
      association 'reservation_for' [ HMS::Design::concrete::HotelEntity(N)(-) HMS::Design::concrete::Reservation(N)(-) HMS::Design::
⊞ Abstract aggregation 'has' [Hotel/Expense] has not been refined.
⊞ Abstract aggregation 'employs' [Employee/Hotel] has not been refined.
⊞ Concrete class 'TransactionCollection' has no corresponding abstraction.
```

Fig. 5. UML/Analyzer tool depicting inconsistencies

commented that each rule has a reliability number. Our approach also uses those numbers to derive an overall estimation of how accurate the abstraction is. For example, in Figure 5 we see that our tool derived a high-level *association* between *Security* and *Hotel* and indicated it to be 90% reliable (<<0.9>>) certain that it is correct, indicating high trustworthiness. Another way of indicating accuracy is in the inconsistency feedback itself. For instance, in Figure 5 we see a warning asserting that *stays_at* has multiple ambiguous interpretations followed by another warning indicating that *is_checked_in* was removed as a viable interpretation of *reservation_for*. These warnings indicate that one should also investigate the surrounding elements due to ambiguity. The accuracy of our approach is improved if (1) transformation is more reliable and (2) more trace information is provided.

Scalability

In terms of scalability we distinguish computational complexity and manual intervention. Comparison, our actual consistency checking activity is very fast (O(n)) since it only requires the one-time traversal of all model elements and a simple comparison. Comparison with ambiguous reasoning is also fast since the maximum bi-partite algorithm is computationally linear with respect to the number of model elements. Naturally, transformation is more complex but its scalability can be improved by reusing previously derived model elements (see [9] for a detailed discussion on abstraction scalability). This is something a pure comparative consistency checking approach could never do. To date we have applied our tool on UML models with up to several thousand model elements without problems in computational complexity. More significant, however, is the minimal amount of manual intervention required to use our approach. For a small problem it is quite feasible to provide sufficient human guidance (e.g., more traces, less/no ambiguities), however, for larger systems it is infeasible to expect complete model specifications. In that respect, our approach has the most significant benefits. We already outlined throughout this paper how partial specifications, ambiguities, and even complex many-to-many mappings can be managed successfully by our approach. In case of larger systems this implies substantial savings in human effort and cost since complete specifications are often very hard if not impossible to generate manually [13].

Change Propagation

Model-based software development has the major disadvantage that changes within views have to be propagated to all other views that might have overlapping information. Our consistency checking technique supports change propagation in that it points out places where views differ. The actual process of updating models, however, must still be performed manually. Here, transformation may be used as an automated means of change propagation (see also [9]).

Shortcomings of Transformation

Our approach relies on good transformation techniques. Especially in context of a less-than-perfect modeling language, such as the UML, the reliability of transformation suffers. Comparison needs to compensate for deficiencies of transformation methods to reduce the number of false positives. For example, our approach conservatively identifies methods of abstracted classes where the true set of methods is a subset of the transformation result. This transformation deficiency can be addressed by comparison checking for a subset of methods instead of the same set. Other deficiencies can be addressed similarly.

6 Related Work

Existing literature uses transformation for consistency checking mostly as a means of converting modeling information into a more precise, formal representation. For instance, VisualSpecs [4] uses transformation to substitute the imprecision of OMT (a language similar to UML) with formal constructs like algebraic specifications followed by analyzing consistency issues in context of that representation; Belkhouche-Lemus [3] follows along the tracks of VisualSpecs in its use of a formal language to substitute statechart and dataflow diagrams; and Van Der Straeten [25] uses description logic to preserve consistency. We also find that formal languages are helpful, however, as this paper demonstrated, we also need transformation methods that "interpret" views in order to reason about ambiguities. Neither of their approaches is capable of doing that. Furthermore, their approaches create the overhead of a third representation.

Grundy et al. took a slightly different approach to transformation in context of consistency checking. In their works on MViews/JViews [14] they investigated consistency between low-level class diagrams and source code by transforming them into a "base model" which is a structured repository. Instead of reasoning about consistency within a formal language, they instead analyze the repository. We adopted their approach but use the standardized UML's meta model as our repository definition. Furthermore, MViews/JViews does not actually interpret models (like the other approaches above), which severely limits their number of detectable inconsistencies.

Viewpoints [15] is another consistency checking approach that uses inconsistency rules which are defined and validated against a formal model base. Their approach, however, emphasizes more "upsteam" modeling techniques; and has not been shown to work on partial and ambiguous specifications. Nevertheless, Viewpoints also extends our work in that it addresses issues like how to resolve inconsistencies or how to live with them; aspects which are considered outside the scope of this paper.

Koskimies et al. [18] and Keller et al. [17] created transformation methods for sequence and state chart diagrams. It is exactly these kinds of transformations we need; in fact, we adopted Koskimies et al.'s approach as part of ours. Both transformation techniques, however, have the drawback that they were never integrated with a consistency checking approach. This limits their techniques for transformation only. Also, as transformation techniques they have the major drawbacks that extensive specifications and/or human intervention are needed while using them. This is due to the inherent differences between state charts and sequence diagrams. Ehrig et al. [11] also emphasizes model transformation. In their case they take collections of object diagrams and reason about their differences. They also map method calls to changes in their object views, allowing them to reason about the impact methods have. Their approach has, however, only been shown to work for a single type of view and they also have also not integrated their approach into a consistency checking framework.

Our work also relates to the field of transformational programming [20,22]. We have proposed a technique that allows systematic and consistent refinement of models that, ultimately, may lead to code. The main differences between transformational programming and our approach are in the degrees of automation and scale. Transformational programming is fully automated, though its applicability has been demonstrated primarily on small, well-defined problems [22]. Our refinement approach, on the other hand, can be characterized only as semi-automated; however, we have applied it on larger problems and a more heterogeneous set of models, typical of real development situations.

SADL [21] follows a different path in formal transformation and consistency. This approach makes use of a proof-carrying formal language that enables consistent refinement without human intervention. The SADL approach is very precise, however, has only been shown to work on their language. It remains unknown whether a more heterogeneous set of models can be also refined via this approach. Also, the SADL approach has only been used for small samples using small refinement steps.

Besides transformation, another key issue of consistency checking is the traceability across modeling artifacts. Traceability is outside the scope of this work but, as this paper has shown, it is very important. Capturing traces is not trivial, as researchers have recognized [13], however, there are techniques that give guidance. Furthermore, process modeling is also outside the scope, although we find it very important in the context of model checking and transformation. To date, we have shown that a high degree of automation is possible, but have not reached full automation yet. Processes are important since they must take over wherever automation ends [19,23].

7 Conclusion

This paper presented a transformation-based consistency checking approach for consistent refinement and abstraction. Our approach separates model validation into the major *Mapping (Traceability)*, *Transformation*, and *Comparison* which may be applied iteratively throughout the software development life cycle to adapt and evolve software systems. To date, our approach has been applied successfully to a number of third party models including the validation of a part of a Satellite Telemetry Processing, Tracking, and Commanding System (TT&C) [2], the Inter-Library Loan System [1] as well as several reverse-engineered tools (including UML/Analyzer itself).

We invented and validated our abstraction technique in collaboration with Rational Software. Our consistency checking approach is fully automated and tool supported. Our approach is also very lightweight since it does not require the use of third-party (formal) languages [4,15,21] but instead integrates seamlessly into existing modeling languages. We demonstrated this in context of the Unified Modeling Language and C2SADEL.

References

1. Abi-Antoun, M., Ho, J., and Kwan, J. Inter-Library Loan Management System: Revised Life-Cycle Architecture. 1999.
2. Alvarado, S.: "An Evaluation of Object Oriented Architecture Models for Satellite Ground Systems," *Proceedings of the 2nd Ground Systems Architecture Workshop (GSAW)*, February 1998.
3. Belkhouche, B. and Lemus, C.: "Multiple View Analysis and Design," *Proceedings of the Viewpoint 96: International Workshop on Multiple Perspectives in Software Development*, October 1996.
4. Cheng, B. H. C., Wang, E. Y., and Bourdeau, R. H.: "A Graphical Environment for Formally Developing Object-Oriented Software," *Proceedings of IEEE International Conference on Tools with AI*, November 1994.
5. Cormen, T.H., Leiserson, C. E., Rivest, R. L.: Introduction to Algorithms. MIT Press, 1996.
6. Egyed A.: A Scenario-Driven Approach to Trace Dependency Analysis. *IEEE Transactions on Software Engineering (TSE)* 29(2), 2003, 116-132.
7. Egyed, A.: "Compositional and Relational Reasoning during Class Abstraction," *Proceedings of the 6th International Conference on the Unified Modeling Language (UML)*, San Francisco, USA, October 2003.
8. Egyed, Alexander. Heterogeneous View Integration and its Automation. PhD Dissertation, Technical Report from the University of Southern California, USA, 2000.
9. Egyed A.: Automated Abstraction of Class Diagrams. *ACM Transaction on Software Engineering and Methodology (TOSEM)* 11(4), 2002, 449-491.
10. Egyed, A. and Medvidovic, N.: "A Formal Approach to Heterogeneous Software Modeling," *Proceedings of 3rd Foundational Aspects of Software Engineering (FASE)*, March 2000, pp.178-192.
11. Ehrig H., Heckel R., Taentzer G., and Engels G.: A Combined Reference Model- and View-Based Approach to System Specification. *International Journal of Software Engineering and Knowledge Engineering* 7(4), 1997, 457-477.
12. Ford, L.R., Fulkerson, D. R.: Flows in Networks. Princeton University Press, 1962.
13. Gieszl, L. R.: "Traceability for Integration," *Proceedings of the 2nd Conference on Systems Integration (ICSI 92)*, 1992, pp.220-228.
14. Grundy J., Hosking J., and Mugridge R.: Inconsistency Management for Multiple-View Software Development Environments. *IEEE Transactions on Software Engineering (TSE)* 24(11), 1998.
15. Hunter, A. and Nuseibeh, B.: "Analysing Inconsistent Specifications," *Proceedings of 3rd International Symposium on Requirements Engineering (RE97)* , January 1997.
16. Jackson, J.: "A Keyphrase Based Traceability Scheme," *IEE Colloquium on Tools and Techniques for Maintaining Traceability During Design*, 1991, pp.2-1-2/4.
17. Khriss, I., Elkoutbi, M., and Keller, R.: "Automating the Synthesis of UML Statechart Diagrams from Multiple Collaboration Diagrams," *Proceedings for the Conference of the Unified Modeling Language*, June 1998, pp.132-147.
18. Koskimies K., Systä T., Tuomi J., and Männistö T.: Automated Support for Modelling OO Software. *IEEE Software*, 1998, 87-94.
19. Lerner, B. S., Sutton, S. M., and Osterweil, L. J.: "Enhancing Design Methods to Support Real Design Processes," *IWSSD-9*, April 1998.

20. Liu, J., Traynor, O., and Krieg-Bruckner, B.: "Knowledge-Based Tranformational Programming," *4th International Conference on Software Engineering and Knowledge Engineering*, 1992.
21. Moriconi M., Qian X., and Riemenschneider R. A.: Correct Architecture Refinement. *IEEE Transactions on Software Engineering* 21(4), 1995, 356-372.
22. Partsch H. and Steinbruggen R.: Program Transformation Systems. *ACM Computing Surveys* 15(3), 1983, 199-236.
23. Perry, D. E.: "Issues in Process Architecture," *9th International Software Process Workshop*, Airlie, VA, October 1994.
24. Rumbaugh, J., Jacobson, I., Booch, G.: The Unified Modeling Language Reference Manual. Addison Wesley, 1999.
25. Van Der Straeten, R., Mens, T., Simmonds, J. and Jonckers, V.: “Using Description Logic to Maintain Consistency between UML Models,” Proceedings of 6th International Conference on the Unified Modeling Language, San Francisco, USA, 2003, pp. 326-340..