

Automatically Validating Model Consistency during Refinement

Alexander F. Egyed

Computer Science Department
University of Southern California
941 W. 37th Place, SAL 328
Los Angeles, CA 90089
+1 213 740 6504
aegyed@sunset.usc.edu

ABSTRACT

Automated consistency checking between software development models still remains a complex and non-scalable problem. Current solutions are frequently only able to detect small numbers of inconsistency types, often under less than realistic assumptions. This paper introduces a new approach to consistency checking based on model transformation. Our approach uses transformation to translate and to interpret model information between different types of views (e.g., diagrams) in order to simplify their comparison. Transformation-based consistency checking, in the manner we use it, has never been attempted before and, as this paper will demonstrate, has significant benefits including (1) increased variety of automatically detectable inconsistencies, (2) improved scalability, (3) ability to handle incomplete, ambiguous model specifications, and (4) ability to define domain- and model-independent inconsistency rules. This paper will illustrate our approach in context of model refinement and abstraction using a complex example. Our approach is fully tool supported.

Keywords

Consistency checking, transformation, class diagrams

1 INTRODUCTION

In the past decades numerous software models were created to support software development at large. Models have usually in common that they break up software development into smaller, more comprehensible pieces utilizing a divide and conquer strategy. Models are extremely useful in that respect since “it is not the number of details, as such, that contributes to complexity, but the number of details of which we have to be aware at the same time [21].”

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 sys-

tem is investigated, each through its own models, then the validity of solutions derived from those models requires that commonalities (redundancies) between them are recognized and maintained in a consistent fashion. Consistency between models, however, cannot easily be guaranteed since models embody information redundancies (information overlap) to enable their closed-world environments. Today, consistency checking between models has to be done mostly manually, resulting in high costs.

In this work, we introduce a transformation-based approach to consistency checking. We discuss our approach in context of abstract and concrete views (views are representations of partial models like diagrams). Our approach is generic enough in that it does not matter what views are created first. Thus, validating the consistency of an abstract view that has been reverse engineering from a concrete view (e.g., source code) can be treated equally to validating the consistency of a concrete view that has been (manually) refined from an abstraction. We will demonstrate our approach in context of UML class diagrams [20]. In previous works, we also demonstrated our approach on UML object diagrams and the C2SADEL language (the latter will not be discussed in detail here since the C2SADEL and UML integration has been published previously [8]). Our transformation-based consistency checking approach adds the following novel contributions:

1. Larger scope of automatically detectable inconsistencies including types of model inconsistencies that were not detectable automatically previously,
2. Improved scalability due to reuse of transformation results and optimized transformation infrastructure,
3. Higher effectiveness in handling incomplete specifications and other types of ambiguities,
4. More generic inconsistency rules applicable to a broader set of models (e.g., some rules applicable to class, object, and state chart diagrams).

To evaluate our transformation-based approach to consistency checking we also applied it successfully to other types of heterogenous models like state chart diagrams and sequence diagrams [5]. 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 large and non-trivial third-party

Technical Report

University of Southern California
Computer Science Department,
941 W. 37th Place, SAL 328,
Los Angeles, CA 90089-0781, USA

Also submitted to the 23rd International Conference on Software Engineering (ICSE 2001)

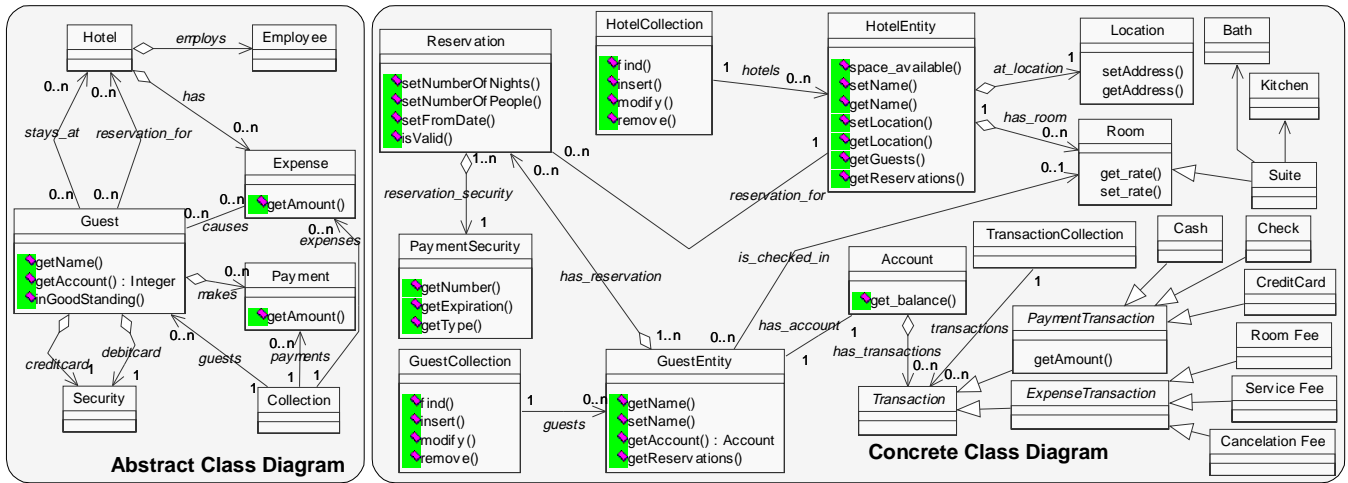


Figure 1. Abstract and concrete class diagrams (entity view) of a hotel management system (HMS)

models [1,2] as well as in-house developed models. Additionally, our approach has been fully implemented enabling us to gather extensive metrics (e.g., computational complexity and scalability). Our tool, called UML/Analyzer, enables consistency checking and model abstraction (transformation) for UML object and class diagrams. The tool also implements all transformation and consistency checking measures discussed in this paper (and more).

The remainder of this paper is organized as follows: Section 2 introduces a complex example and discusses abstraction and refinement problems in context of two class diagrams depicted there. In Section 3, we will highlight consistency checking without transformation and discuss in what cases it is effective and in what cases not. Section 4 introduces one of our transformation methods and then discusses how this method 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 works.

2 EXAMPLE

Figure 1 depicts two class diagrams showing the entities of a *Hotel Management System* (HMS) at two levels of abstraction. The left diagram is the more abstract class diagram with abstract classes like *Guest* and *Hotel* and relationships like “a guest may either stay at a hotel or may have reservations for it.” The abstract 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 (e.g., in form on a credit card). The diagram uses three types of relationships to indicate uni-directional, bi-directional, and part-of dependencies (see UML definition [20]). 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 right side of Figure 1 depicts a refinement of the left side. It can be seen that the basic entities like *Guest* or *Expense* are still present although named slightly differently¹. It can also be seen that additional classes were introduced. For instance, the concrete diagram makes use of new classes like *Reservation* or *Check* to refine or extend the abstract diagram. The concrete (refined) class diagram also uses the same types of relationships as the abstract one plus generalization relationships (triangle head) to indicate inheritance. The concrete diagram also describes methods associated with classes more extensively.

3 SIMPLE CONSISTENCY CHECKING

Having two diagrams depicting the same information at different levels of abstraction implies that there must be information overlap between them. Both diagrams describe the same part of the HMS system separately and, in doing so, they also share modeling information about their common environment. For instance, we can see that the abstract class *Guest* was refined into the concrete class *GuestEntity* or that the abstract relationship “has” (between *Guest* and *Expense*) was refined and extended into multiple relationships with the new classes *Transaction* and *Account* between them. Note that the knowledge on how model information in different views relate to one another is commonly referred to as traceability (or mapping) [10].

Current consistency checking approaches detect inconsistencies by traversing models and validating those models against inconsistency rules. For instance, consistency checking approaches like JViews (MViews) [11], View-Points [12] or VisualSpecs [4] read diagrams, translate² them into common (and usually formal) representation schemes, and validate inconsistency rules against them. These approaches, although successful in some cases, make three fundamental and non-realistic assumptions: First, they assume that the traceability among modeling information

¹ 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.

² There, translation is sometimes referred to as transformation; however, we have a different meaning for that term.

(the knowledge on how diagrams relate) is fully known, second, they assume that that traceability follows a simple one-to-one mapping; and third, building on the first two assumptions, they assume that inconsistencies can be generally identified via direct comparisons where the structure and extend of the comparison is unambiguous.

In the past three years, we have inspected a large number of third-party software development models [1] and observed that above three assumptions rarely ever hold. In order to demonstrate how our approach stands up against such simplifications, we chose an example that violates all of them, making our example more complex and realistic. To violate the first assumption, we provide only a small set of traceability information between the diagrams:

Table 1. Mapping (traceability) data

Abstract Diagram	Concrete Diagram
<i>Guest</i>	<i>GuestEntity</i>
<i>Hotel</i>	<i>HotelEntity, Room, HotelCollection</i>
<i>Security</i>	<i>PaymentSecurity</i>
<i>Expense</i>	<i>ExpenseTransaction</i> and <i>Room Fee</i>
<i>Payment</i>	<i>PaymentTransaction</i>
<i>Collection</i>	<i>GuestCollection, and HotelCollection</i>

Because of the limited set of traceability information, the example in Figure 1 also violates the second assumption about the one-to-one mapping. We now find several cases of one-to-many mappings (e.g., *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 concrete diagram), one case of a many-to-one mapping (*HotelCollection* is assigned to *Hotel* and *Collection*), and many cases of no mapping altogether (e.g., *Employee* in the abstract diagram or *Account* in the concrete diagram).

Finally, we also violate the third assumption above in that a direct comparison is now mostly impossible or the comparison is ambiguous. For instance, a case of ambiguity is reflected in the many-to-many mapping of *reservation_for* and *stays_at* to the concrete diagram where existing modeling information is not sufficient to infer what their refinements are. A human observer may still guess that the relationship *reservation_for* maps to the class *Reservation* and its single relationships to *Guest* and *Hotel*, however, due to the lack of traceability information, a machine cannot easily come to the same conclusion.³ Our example also has several cases where direct comparison is impossible. For instance, the abstract relationship from *Guest* to *Payment* has no direct (one-to-one) counterpart in the concrete diagram. As it can be seen, there is no direct relationship between *GuestEntity* and *PaymentTransaction*, its concrete counterparts, but instead that relationship is clouded through the use of “helper” classes (e.g., *Account* and *Transaction*) that refine the abstract relationship.

³ Although we could employ a synonym checker to duplicate the human reasoning, such an approach would still be flawed.

To summarize, our example in Figure 1 exhibits a series of non-trivial and complex features. By no means, is our example a “special case” which requires special attention. In fact, these complexities are representative of the complexities in the entire HMS and many other software models we inspected. To date there exists no consistency checking approach that can perform *exhaustive* consistency validations on such an example. The next section will show how our approach addresses these challenges.

4 TRANSFORMATION-BASED CONSISTENCY CHECKING

The example in Figure 1 exhibits multiple problems that will be addressed in different parts of this section. The two primary challenges are (1) how to enable comparison between model elements that do not have a simple one-to-one mapping and (2) how to compare model elements that are ambiguously mapped.

View Integration Framework

To identify inconsistencies in an automatable fashion, we have devised and applied a view integration framework, accompanied by a set of activities and techniques [5]. Our view integration approach exploits the redundancy between views: for instance, since the abstract diagram in Figure 1 contains information also used in the concrete diagram, this information can be seen as a constraint between the abstract and concrete diagrams. Our view integration framework enforces such constraints and, thereby, the consistency across these two views. In addition to constraints and consistency rules, our view integration framework also defines *what* information can be exchanged and *how* information can be exchanged. This is critical for scalability and automateability. Our approach has the following activities:

- **Mapping:** identifies and cross-references related modeling elements that describe overlapping and thus redundant pieces of information. Mapping is often done manually via naming dictionaries or traceability matrices (e.g., see trace matrix in Table 1). Mapping assists consistency checking by defining *what* to compare.
- **Transformation:** converts modeling elements or diagrams into intermediate models in such a manner that they (or pieces of them) can be understood easier in the context of other diagram(s). Transformation assists consistency checking by defining *how* to compare.
- **Differentiation:** compares model elements and diagrams with intermediate models that were generated through transformation where differences indicate inconsistencies.

Figure 2 depicts the use of *Transformation, Mapping* and *Differentiation* in context of our example. The figure shows two basic inconsistency detection approaches. It shows that in order to compare the two user-defined⁴ views A and C (for abstract and concrete), we could either (a) compare them directly or (b) transform C into ‘something like A’ so that C becomes easier comparable to A. There are addi-

⁴ User-defined views are diagrams that are created by humans (e.g., Figure 1). Derived views (interpretations) are diagrams that are automatically generated via *Transformation*.

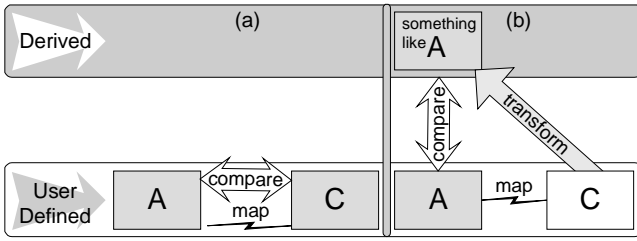


Figure 2. View transformation and mapping to complement view comparison (differentiation)

tional consistency checking approaches not depicted in Figure 2, like (c) to transform A into ‘something like C’ so that A becomes easier comparable to C or (d) to transform both A and C into a third type so that they become easier comparable in the context of that third type. Those latter approaches are, however, not used in this paper.

We already discussed the shortcoming of approach (a), however, listed it here again since it describes a legitimate consistency checking approach that works in some cases [4,11,12]. Approach (b) is the one we have taken in this paper. This approach transforms (abstracts) the concrete diagram into a form that makes the result directly comparable with the abstract diagram. As part of our collaboration with Rational Corporation we created an abstraction technique [7] and, in this paper, we will demonstrate its integration with our consistency checking technique.

For completeness, the third other approach (c) mentioned above suggests the refinement of the architecture into a design so that the designs can be compared. This approach is not possible here since there currently are no fully automated refinement techniques available. Finally, the fourth approach (d) suggests transforming both diagrams into a third representation scheme that allows their direct comparison. The example in this paper does again not require such an approach, however, it must also be noted that, although the latter two approaches are not used here, we still encountered cases where they are needed while integrating other types of diagrams. For instance, we have also integrated class/object diagrams with the C2SADEL architecture description language where approach (d) was needed to first establish a “common” repository [8]

Abstraction implementing Transformation

In the course of evaluating nine types of software models [5] (class, object, sequence, and state chart diagrams, their abstractions and C2SADEL) we identified the need for four transformation types called *Abstraction*, *Generalization*, *Structuralization*, and *Translation*. This paper focuses on inconsistencies during refinement and thus only needs *Abstraction*. See [5] for a discussion of the other types.

Abstraction deals with the simplification of information by removing details not necessary on a higher, more abstract level. We distinguish between basically two types of abstraction techniques called *classifier abstraction* and *relationship abstraction*, both of which are based on diagrammatic views using box-and-arrow type representations (e.g., class diagrams or state chart diagrams). Classifier abstraction is probably the more intuitive abstraction type since it

closely resembles hierarchical decomposition of systems provided in many views. For instance, in UML, layers 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, forming a tree-like hierarchy. In relation abstraction it is the relations (arrows) and not the classes (boxes) that serve as vehicles for abstraction. Relations (with classes) can be collapsed into more abstract relations. Relation abstraction is needed since it is frequently not possible to maintain a strict hierarchy of classes. Since our abstraction technique has been published previously, we will only provide a brief summary here. For a more detailed discussion, please refer to [6].

In order to abstract the concrete diagram in Figure 1, we have to apply both abstraction types. Figure 3 shows a partial view of Figure 1 depicting, in the top layer, the abstract classes *Hotel*, *Guest*, and *Payment* and, in the bottom layer, their concrete counterparts *HotelEntity*, *HotelCollection*, *Room*, *GuestEntity*, and *PaymentTransaction* (recall Table 1). The bottom layer also depicts all relationship paths between these concrete classes, i.e., like the path that originates from *GuestEntity* with an aggregation to *Reservation* followed by an association to *HotelEntity*.

The first abstraction step groups concrete classes that belong to single abstract classes. For instance, the concrete classes *HotelEntity*, *HotelCollection*, and *Room* are all part of the same abstract class *Hotel* (Table 1). Classifier abstraction is used to group them and create a more abstract, derived class called *Hotel*. That derived class is depicted in the first (1) derived view in Figure 3. Besides grouping the three concrete classes, the abstraction method also replicated the inter-dependencies of those three classes into the derived, abstract class. It can be seen that the derived class *Hotel* now has relationships to *Reservation* and *Guest* that were “inherited” from *HotelEntity* and *Room* respectively. Also note that the single concrete classes *GuestEntity* and *PaymentTransaction* were grouped into the more abstract, derived classes *Guest* and *Payment*. They also inherited all inter-relationships from their concrete counterparts. Classifier abstraction therefore simplified our understanding of the concrete relationships between *Hotel*, *Guest*, and *Payment*. Previously we were not able to reason about the direct interdependencies between concrete classes, however, now we have created one direct relationship between *Guest* and *Hotel*. Still, in order to enable more meaningful consistency checking we also need to eliminate the other helper classes *Reservation*, *Account*, and *Transaction* since they still obstruct our understanding. The problem is that those classes were not assigned to any abstract classes, thus, cannot be eliminated via classifier abstraction.

The second abstraction step groups concrete relationships into single abstract relationships. For instance, the concrete relationship path going from *Guest* via *Reservation* to *Hotel* in Figure 3 (bottom) describes a semantic dependency between those classes. That dependency, although more elaborate, still has a meaning that can be approximated through simpler, more abstract model elements. In particular, this example shows an aggregation relationship be-

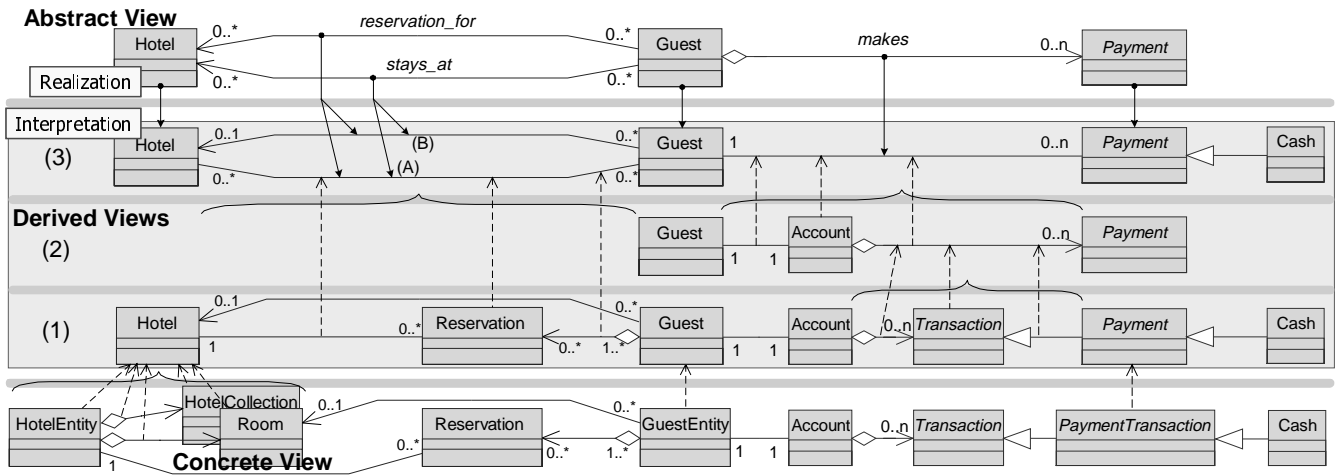


Figure 3. Abstraction applied on part of HMS showing abstract, concrete, and derived modeling information⁵

Table 2. Excerpt of abstraction rules for classes [6]

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

tween 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 abstract) 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 relationship of type *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 fifty class abstraction rules. A large set of abstraction rules for UML class diagrams is published in [6]. Table 2 shows a small sample of abstraction rules need 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. Fur-

thermore, 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).

Classifier and relationship abstraction must also be applied to the rest of the concrete diagram (Figure 1). Consequently, abstraction simplifies our understanding of the transitive (indirect) relationships between important classes. Once abstraction is completed, all helper classes have been eliminated leaving behind their abstract *interpretations*. This abstract interpretation is similar in structure to the user-defined abstract class diagram (also compare top two layers of Figure 3) and both can now be compared (recall Figure 2b). Obviously, our abstraction method satisfies our need as a transformation method in that it truly simplifies the relationships between the user-defined abstract and concrete diagrams. Arrows with circular arrowheads on one end are generated automatically as part of the abstraction process to indicate comparable model elements. These arrows are trace information (*interpretation traces*) that, together with the *abstraction traces* (dashed arrows with regular arrow heads), forms the foundation for consistency checking as discussed next.

Differentiation

In the beginning of Section 4 we listed two primary challenges in dealing with incomplete model information. The previous subsection solved the first one by showing how partial mappings (traces) can still be used to meaningfully abstract class diagrams. This section deals with how to identify inconsistencies, also addressing the second challenge in how to deal with ambiguities. In the following, we

⁵ In order to improve the readability of the figure, we decided to duplicate model elements (e.g., *Account*). In reality, data is kept in what we call a *reduced redundancy model* which almost completely eliminates redundancy. To display it here would make the figure unreadable. Furthermore is the discussion of the reduced redundancy model outside the scope of this paper.

will first describe the basics of our consistency-checking approach (differentiation) under normal conditions, followed by an explanation how ambiguities are treated. The *Differentiation* component of our framework uses consistency rules and validates them against our model repository. In Figure 3, the third derived view represents the information the concrete view (bottom layer) has in common with the abstract view (top layer). *Differentiation* is used to detect differences between the derived and abstract views, where differences are general indications of inconsistencies. We use the term “indication” to point out that detected inconsistencies must be treated as potentially inconsistent since any form of automated model analysis has the potential of errors.

In the following we present a small sample of consistency rules that built upon the derived views generated in Figure 3. 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 true⁶:

1. Type of concrete relation is different from abstraction:
 $\forall r \in \text{relations}, \text{size}(r \rightarrow \text{interpretations}) > 0 \Rightarrow$
 $[\exists i \in r \rightarrow \text{interpretations}, \text{type}(i) = \text{type}(r)]$

Rule 1 states that for two relations to be consistent they must also have the same types. Its qualifier (before “ \Rightarrow ”) defines that this rule only applies to relations that have interpretations (interpretations are trace links that indicate comparable data; indicated via arrow with circular arrow head in Figure 3). In Figure 3, we have six interpretation traces; three of which are originating from relationships (circular end denotes point of origin). Above rule, then states that there must be at least one interpretation of that relation for which the type of that relation is equal to the type of its interpretation. In Figure 3, the abstract relations “stays_at” and “reservation_for” satisfy above condition⁷, however, the abstract relation “makes” does not. This case denotes an inconsistency since “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), we are able to identify the concrete user-defined model elements (e.g., classes *Account* and *Transaction* as well as their relationships to *GuestEntity* and *PaymentTransaction*) that contributed to the inconsistent, derived interpretation.

2. Concrete relation has no corresponding abstraction:
 $\forall r \in \text{relations}, \text{size}(r \rightarrow \text{abstractions}) = 0 \wedge$
 $\text{size}(r \rightarrow \text{realizations}) = 0 \Rightarrow$
 $\neg [\exists c \in r \rightarrow \text{classifiers}, \text{size}(c \rightarrow \text{realizations}) > 0]$

The first rule showed a case on how to compare model elements in case interpretation arrows exist. Rule 2 shows that there are also variations. In particular, it is defined here that all concrete relations must somehow be traced to abstract model elements; if such a trace is missing then the

⁶ Some qualifier conditions were omitted for brevity (e.g., checking for transformation type) since they are not needed here.

⁷ 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.

concrete relation may be outside the scope of the abstract view. 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* (realization traces are the opposite ends of interpretations traces). Figure 3 has many relations (derived and user-defined ones). Checking for relations that do not have abstractions ensures that only the most abstract relations are compared. This eliminates an erroneous inconsistency since concrete relations (e.g., the aggregation from *Transaction* to *Account*) are not considered because they are part of the creation of derived interpretations. Checking for relations that additionally do not have realizations eliminates relations such as the derived association between *Guest* and *Payment* since it is related to the abstract view, indicating that it is “within” the scope of the abstract one. There are still several relations left (e.g., generalization from concrete *Cash* to *Payment* or aggregation from abstract *Guest* to *Payment*). The rule therefore continuous on and defines that consistency is ensured if *none* of the classes the relations point to have realizations either. That last part of the rule makes sure that abstract relations (e.g., abstract view) are not listed as inconsistencies. There remains only one relation in Figure 3 that violates this rule; the generalization from *Cash* to *Guest*. This generalization has neither an abstraction nor a realization trace; however, its neighbor class *PaymentTransaction* has one via the “derived” *Payment*⁸.

3. Destination direction/navigability of concrete relation does not match abstract relation:
 $\forall r \in \text{relations}, \text{size}(r \rightarrow \text{interpretations}) > 0 \Rightarrow$
 $[\exists i \in r \rightarrow \text{interpretations}, \text{type}(i) = \text{type}(r) \Rightarrow$
 $[\text{size}(r \rightarrow \text{destinationClassifiers} +$
 $i \rightarrow \text{destinationClassifiers} \rightarrow \text{realizations}) = 0]]$

Rule 3 defines that for two relations to be consistent they ought to be pointing in the same directions (same classes). Like the first rule, this one only applies to relations that have interpretations. It states that there must be at least one interpretation with the same type as the realization, were 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 by the qualifier saying that this rule applies to relations of the same type).

In case of Rule 3 and the *reservation_for* and *stays_at* relations we encounter an ambiguous case in that the two (realization) relations point to the same two interpretations (labeled (A) and (B) in Figure 3). The problem is due to the lack of traceability information which causes a dilemma in that it becomes hard to decide what model elements relate to one another. Our approach faces ambiguities under the hypothesis that there ought to be one case that must be consistent. Thus, our approach sequentially traverses all inter-

⁸ 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.”

pretations that may apply and tries to assess each interpretation's consistency with its realization. For instance, in case of the realization *reservation_for*, our framework compares it with both derived interpretations (labeled (A) and (B) in Figure 3) individually. Applying our previous consistency rule on the derived interpretation (A), it then finds that it is inconsistent with the abstract relation *reservation_for* since the directions of their arrowheads do not fully match. It thus eliminates the derived relation (A) as a viable interpretation of *reservation_for*. Our approach next compares *reservation_for* with the derived interpretation (B), for which it finds a consistent use of arrowheads. The relation *reservation_for* thus “force” the interpretation (B) as being its only, unambiguous interpretation.

Ambiguous Reasoning

In case multiple interpretations were found to be consistent or inconsistent, the ambiguity would have remained until (1) other model elements “force” away some of its interpretations or (2) other consistency rules are found that “filter” away interpretations. The effect of that can be seen in the case of the abstract relation *stays_at*. When it is validated for directional consistency (rule 3), it is initially diagnosed as being also ambiguous (again note the interpretation trace in Figure 3). However, prior to its validation, a check would be performed, determining whether any one of *stays_at*'s interpretations was validated previously, and if yes, whether any one of them were taken away (by force). In this case, we indeed find that *reservation_for* has taken away interpretation (B), leaving *stays_at* with only one, thus unambiguous option of validating its consistency with interpretation (A) for which it determines 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. For instance, if *stays_at* had been validated before *reservation_for*, it would have forced away interpretation (B) for itself, leaving *reservation_for* with an inconsistent interpretation (A). Nevertheless, in both scenarios the system would find the same inconsistency, plus also generate a warning about the existence of the ambiguities pointing in the proper direction. We see this case analogous to compilers where sometimes compiler-generated error messages do not exactly point to the real error but instead to elements close to them (usually statements thereafter). Thus, inconsistency feedback has to be interpreted cautiously. This case, like any other case, requires a human judgment call at the end.

To generalize from the previous rules and examples, in validating consistency among (concrete/abstract) model elements we potentially encounter five situations as depicted in Figure 4. Situation (a) is the most simplistic one where there is a one-to-one mapping between interpretation

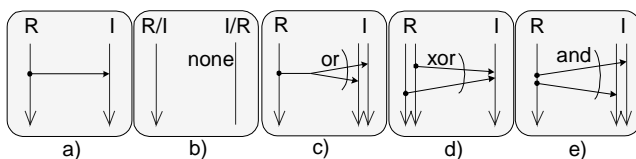


Figure 4. Basic Differentiation Rules for Ambiguity

(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 concrete interpretation (concrete relation) that had no abstraction. The reverse is also possible where there is an abstract realization that has no refinement. While discussing Rule 3 with its ambiguity example, we encountered situation c) where one realization has two or more interpretations (one-to-many mapping). This scenario required the validation of the consistency of at least one interpretation (OR condition) for overall consistency to be true. Scenario d) was not discussed thus far; however, Figure 1 shows an example. There, in the abstract view, we find that *Guest* has two different aggregations (*debitcard* and *creditcard*) to *Security*. Upon abstracting the concrete view we indeed derive an aggregation interpretation between *Guest* and *Security*, however, only one (many-to-one mapping). In this case, only one of the two abstract aggregations (realizations) is allowed to “force” the derived interpretation away (XOR condition), leaving the other aggregation with no interpretation (scenario b). We also used scenario d) in connection with scenario c) previously to resolve an ambiguity case (many-to-many mapping) between *reservation_for* and *stays_at* and its interpretations, which shows, that more complex many-to-many problems can be reduced to a sequence of basic consistency checking scenarios. Finally, scenario e) shows a case where multiple distinct interpretations exist for a single realization. Note that this case is different from scenario c) in that separate transformations created those interpretations. For instance, we could derive an interpretation of *reservation_for* through abstraction of a concrete class diagram but also through generalization of a specific sequence diagram (showing a kind of test scenario), or through the structuralization of a behavioral state chart diagram [5]. In those cases, we might identify multiple, distinct interpretations for *reservation_for* and all of them have to be consistent (AND condition). Again, scenario e) can also be combined with other scenarios to enable more complex, heterogeneous consistency checking [5].

Generality of Consistency Rules

Earlier in the paper we claimed our rules to be very generic. Let us recall rule 1 where we compared the type of a concrete relation with its abstract counterpart.

4. Type of concrete classifier is different from abstraction:

$$\forall c \in \text{classifiers}, \text{size}(c \rightarrow \text{interpretations}) > 0 \Rightarrow [\exists i \in c \rightarrow \text{interpretations}, \text{type}(i) = \text{type}(c)]$$

Rule 4 does the same for classes. As it can be noted, both consistency rules look almost alike in structure. Actually, we could merge them by replacing “ $c \in \text{classifiers}$ ” with “ $c \in \text{modelelements}$.” In fact, that same consistency rule can then also be used to validate the consistency between concrete and abstract state chart diagrams, making this rules generic across box-and-arrow-type, heterogeneous views (it is not done here to enable a finer granularity during inconsistency detection). There are, however, exceptions to above generality. Consider rule 5 which validates the consistency of cardinality (multiplicity) between relations (e.g., “0..n” or “1” in Figure 1):

5. Cardinality of concrete relation different from abstraction

$$\forall r \in \text{relations}, \text{size}(r \rightarrow \text{interpretations}) > 0 \Rightarrow$$

$$[\exists i \in r \rightarrow \text{interpretations}, \text{type}(i) = \text{type}(r) \wedge$$

$$\text{size}(r \rightarrow \text{destinationClassifiers} +$$

$$i \rightarrow \text{destinationClassifiers} \rightarrow \text{realizations}) = 0 \wedge$$

$$[r \rightarrow \text{cardinality} = i \rightarrow \text{cardinality}]]$$

In UML, only relations of class diagrams use cardinality. Thus it is neither meaningful nor possible to generalize this rule to, say, state transition links (relations in statechart diagrams) nor is it meaningful to apply this rule onto classes. This restriction, which limits the scope of generality of our consistency rules, is not caused by our consistency checking framework but instead by the domain. Also note that rule 5 has an extensive qualifier condition. The rule states that cardinality checking is only useful if the interpretation and realization have the same type and the same destination classes. The latter condition is important since knowing about consistent cardinalities to inconsistent classes has no value or purpose. Also note that “+” denotes a symmetric difference (XOR for sets).

5 DISCUSSION

Scope

Additionally to the five (in)consistency rules presented in this paper, we identified almost 20 more that apply to refinement [5]. Figure 5 (bottom left) shows an excerpt of the list of inconsistencies between the diagrams in Figure 1 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 (right) of the concrete diagram (Figure 1). 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 concrete model elements (e.g., *Reservation*) as the potential cause of the inconsistency. The usefulness of our transformation-based consistency checking approach is neither limited to class

diagrams nor limited to the UML. We also identified around 40 additional inconsistency types between other types of UML diagrams [5] (sequence and state chart diagrams) as well as the non-UML language C2SADEL [8]. In these cases, transformation-based consistency checking was used, like discussed in this paper.

Accuracy (True Inconsistencies/False Inconsistencies)

An important factor on how to estimate the accuracy of our approach (or any consistency checking approach) is in measuring how often it provides erroneous feedback (e.g., report of inconsistencies were there are none). 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 2 we presented abstraction rules and 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 an abstract *association* between *Security* and *Hotel* and indicated that it is 90% (<<0.9>>) certain that it is correct, indicating high trustworthiness.

Another way how accuracy can be estimated is in the inconsistency feedback itself. For instance, in Figure 5 we see a warning asserting that *stays_at* has multiple ambiguous interpretations. Later on we encounter 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.

There are two ways on how to improve the accuracy of our approach: (1) provide more reliable (detailed) abstraction rules and (2) provide more trace information. For instance, if Table 1 had contained a trace from the concrete class *Reservation* to the abstract relation *reservation_for* then the whole ambiguity with respect to the relationships between *Guest* and *Hotel* would have been avoided.

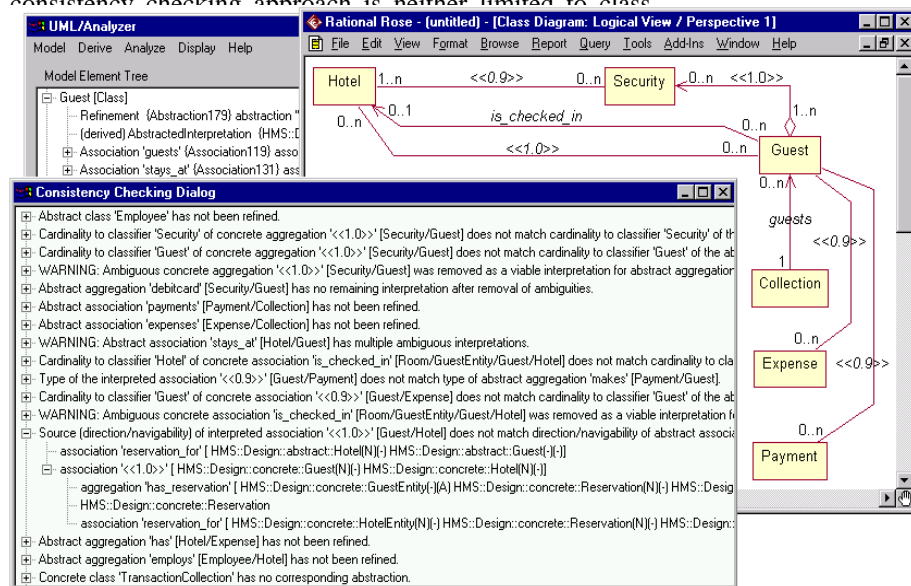


Figure 5. UML/Analyzer tool depicting inconsistencies

Scalability

In terms of scalability we distinguish computational complexity and manual intervention. Differentiation, 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. Naturally, transformation is more complex but its scalability is improved due to the reuse of derived model information (something a pure comparative consistency checking approach could never do [4,11,12]). We already encountered the benefits of reuse previously. For instance, when we abstracted the path from *PaymentTransaction* to *GuestEntity* via *Account* and *Transaction*, the path *GuestEn-*

tity/Account/Transaction needed to be investigated as a sub-path. Later, when the path from *ExpenseTransaction* to *GuestEntity* was abstracted, we found that it shared the same sub-path *GuestEntity/Account/Transaction*. The second abstraction was thus able to reuse results of a previous abstraction.

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 successfully managed 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 [10].

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. Our transformation technique, on the other hand, may also be used as an automated means of change propagation (see also [6]).

Due to the extensive use of transformation, as part of consistency checking, we encounter another change propagation issue. Since derived interpretations are generated out of user-defined views, those interpretations may also be affected by changes within user defined views. For instance, in Figure 3 the concrete class *Transaction* is used to generate the derived association between *Guest* and *Payment* (see layer 3). As we discussed previously, scalability requires that derived information is not discarded, creating the predicament that derived information may become outdated if user-defined information changes. For instance, if the class *Transaction* is deleted, the derived association between *Guest* and *Payment* also becomes obsolete.

In order to remedy this problem we apply the concept of *lazy transformation* and *purging*. *Purging* is used to traverse derived model elements when a change happens. For instance, if the class *Transaction* were deleted, purging would follow all its abstraction traces (Figure 3) to find elements that were derived from it and delete them as well. Since a derived element in turn may have been the foundation for other derivations, purging happens recursively. *Lazy transformation* then ensures that derived information is not re-transformed right away. The “lazy” part is important in order to avoid unnecessary purging and re-transformation during model synthesis.

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. 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 [11] 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 [12] is another consistency checking approach that uses inconsistency rules which are defined and validated against a formal model base. Their approach, however, emphasizes more “upstream” 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. [14] and Keller et al. [13] 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. [9] 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 not integrated their approach into a consistency checking framework.

Our work also relates to the field of transformational programming [16,18]. We have proposed a technique that al-

lows 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 [18]. 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 [17] 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 [10], 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 [15,19].

7 CONCLUSION

This paper presented a transformation-based consistency checking approach for consistent refinement and abstraction. Our approach breaks down model checking into the major activities *Mapping*, *Transformation*, and *Differentiation* which may be applied iteratively throughout the software development life cycle. 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 relation abstraction technique in collaboration with Rational Software [7] and have since enhanced it as well as integrated it into our framework. Our consistency checking approach is fully automated and tool supported and our approach distinguishes itself from related works [3,4,9,11-14,16-18] in that it (1) detects a larger variety of inconsistencies, (2) applies equally to forward- and reverse engineering, (3) is much less restrictive in the amount of prior (manual) specification required, (4) can handle ambiguities, and (5) scales to large models. Our approach is also very lightweight since it does not require the knowledge or use of third-party (formal) languages [4,12,17] but instead integrates seamlessly into existing modeling languages. We successfully demon-

strated this in context of the Unified Modeling Language and C2SADEL.

ACKNOWLEDGEMENTS

We wish to gratefully acknowledge Barry Boehm, Rich Hilliard, Philippe Kruchten, and Nenad Medvidovic.

REFERENCES

1. Abi-Antoun, M., Ho, J., and Kwan, J. Inter-Library Loan Management System: Revised Life-Cycle Architecture. Center for Software Engineering, University of Southern California, Los Angeles, CA 90089-0781, USA.
2. Alvarado, S.: "An Evaluation of Object Oriented Architecture Models for Satellite Ground Systems," *Proceedings of the 2nd Ground Systems Architecture Workshop*, February 1998.
3. Belkhouche, B. and Lemus, C.: "Multiple View Analysis and Design," *Proceedings of the Viewpoint 96: 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. Egyed, A.: "Heterogeneous View Integration and its Automation," PhD Dissertation, University of Southern California, Los Angeles, CA, August 2000.
6. Egyed, A.: "Semantic Abstraction Rules for Class Diagrams," *Proceedings of the 15th International Conference of Automated Software Engineering*, Grenoble, France, Sep. 2000.
7. Egyed, A. and Kruchten, P.: "Rose/Architect: a tool to visualize architecture," *Proceedings of the 32nd Hawaii International Conference on System Sciences (HICSS)*, January 1999.
8. Egyed, A. and Medvidovic, N.: "A Formal Approach to Heterogeneous Software Modeling," *Proceedings of 3rd Foundational Aspects of Software Engineering (FASE)*, March 2000.
9. 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.
10. Gieszl, L. R.: "Traceability for Integration," *Proceedings of the 2nd Conference on Systems Integration*, 1992, pp.220.
11. Grundy J., Hosking J., and Mugridge R.: Inconsistency Management for Multiple-View Software Development. *IEEE Transactions on Software Engineering (TSE)* 24(11), 1998.
12. Hunter, A. and Nuseibeh, B.: "Analysing Inconsistent Specifications," *Proceedings of 3rd International Symposium on Requirements Engineering (RE97)*, January 1997.
13. Khriess, 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.
14. Koskimies K., Systä T., Tuomi J., and Männistö T.: Automated Support for Modelling OO Software. *IEEE Software*, 1998, 87-94.
15. Lerner, B. S., Sutton, S. M., and Osterweil, L. J.: "Enhancing Design Methods to Support Real Design Processes," *IWSSD-9*, April 1998.
16. Liu, J., Traynor, O., and Krieg-Bruckner, B.: "Knowledge-Based Transformational Programming," *4th Conference on Software Engineering and Knowledge Engineering*, 1992.
17. Moriconi M., Qian X., and Riemenschneider R. A.: Correct Architecture Refinement. *IEEE Transactions on Software Engineering* 21(4), 1995, 356-372.
18. Partsch H. and Steinbruggen R.: Program Transformation Systems. *ACM Computing Surveys* 15(3), 1983, 199-236.
19. Perry, D. E.: "Issues in Process Architecture," *9th International Software Process Workshop*, VA, October 1994.
20. Rumbaugh, J., Jacobson, I., Booch, G.: The Unified Modeling Language Reference Manual. Addison Wesley, 1999.
21. Siegfried, S.: Understanding Object-Oriented Software Engineering. IEEE Press, 1996.