

Semantic Abstraction Rules for Class Diagrams

Alexander Egyed

*University of Southern California
Computer Science Department
941 W. 37th Place, SAL 327
Los Angeles, CA 90089-0781
aegyed@sunset.usc.edu*

Abstract

When dealing with object-oriented models like class and object diagrams, designers easily get overwhelmed by large amounts of model elements and their interdependencies. To deal with the complexities of large-scale software models, this paper presents rules and methods for automated abstraction. Our approach is tool supported and allows designers to periodically “zoom out” of a model to investigate and reason about its bigger picture. Our technique has also proven to be well suited for consistency checking and reverse engineering.

1. Introduction

Software models (like class diagrams) simplify software development and reduce their complexity by dividing up problems into smaller, more comprehensible pieces. Those smaller problems can then be resolved more easily on an individual basis. On the downside, as software development progresses, models become increasingly large and bulky. This has the effect that it becomes very difficult for designers and developers to inspect and modify those models due to the potentially many interdependencies that may exist.

We have investigated the issue of synthesis and analysis between models and have created mechanisms for automating them [1]. We have found that software designers are in need of connecting models to allow navigation between them but also to allow their inspection and comprehension. In this work, we describe a technique for model abstraction applicable to class and object diagrams. Our technique bridges the gap between concrete and abstract models and serves two major uses:

- Reverse Engineering: our technique can also be used to reverse engineer high-level models out of more concrete (lower-level) models, and
- Consistency Checking: our technique can be used to enable consistency checking between existing concrete and abstract models [2].

Since all these uses may occur frequently during a software development process, there is a need for automation. We have thus created tool support for our rule-based abstraction technique, called UML/Analyzer. The tool currently supports the abstraction of UML-style (Unified Modeling Language [3]) class and object diagrams. Rational Corporation has adopted our technique and has implemented it into a tool called Rose/Architect [4] (also in the context of UML). The usefulness of this technique is neither limited to class and object diagrams, nor is it limited to UML. Our work can be applied to additional UML diagrams such as statechart and collaboration diagrams as well as to non-UML diagrams such as the C2 architectural style (see [1]).

2. Abstraction Mechanism

The process of 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 – *classifier abstraction* and *relation(ship) abstraction*. Both types of abstractions are based on diagrammatic views that use box-and-arrow representations (e.g., class diagrams.). Whereas classifier abstraction is the more intuitive type of grouping model elements, relation abstraction has been widely ignored.

2.1. Classifier Abstraction

Classifier abstraction is more intuitive since it resembles hierarchical decomposition of structures provided in many views. For instance, in UML, layers of packages can be built using a feature of packages that allows them to contain other packages. Thus, a package can be subdivided into other packages, forming a tree hierarchy. In classifier abstraction it are classifiers (e.g., packages, class, states, etc.) that can be grouped (“collapsed”) to yield a more simplified (ergo abstract) view. The relationships of the collapsed, concrete classes then become part of the interface of the more abstract one.

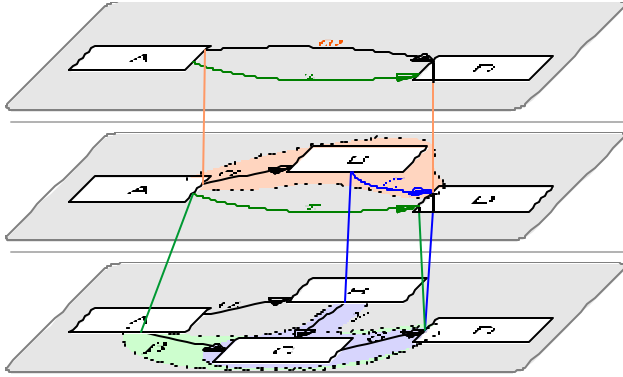


Figure 1: Relation Abstraction

2.2. Relation Abstraction

In relation abstraction [4] it is the relation and not the classifier that serves as a vehicle for abstraction. Relations (with classifiers) can be collapsed into more abstract relations. Figure 1 shows an example of a relation abstraction using three levels of class diagrams. The most concrete level (bottom) contains four classes A, B, C, and D as well as four relations α , β , γ , and δ . The $\beta \rightarrow C \rightarrow \delta$ pattern is collapsed in the middle layer by introducing the composite relation τ . Similarly, $\gamma \rightarrow C \rightarrow \delta$ is collapsed into the composite relation ζ . The third level takes α and B from the lower level as well as ζ from the middle level and further collapses them into the composite relation ω .

The last abstraction showed that composite relations may themselves contain other composite relations. Like classifier hierarchies, composite relations form a tree-like structure (decomposition) between the levels. Circular dependencies between composite elements (both classifiers and relations) are not allowed. For instance, ζ is not allowed to be an abstraction of ω .

Relation abstraction, in context of class and object diagrams, has two major advantages over classifier abstractions: (1) it is useful whenever it is not possible to maintain a strict hierarchy of classifiers, and (2) the diverse types of class relations allow semantic abstractions richer than a pure classifier-based decomposition.

In case of abstracted composite relations (e.g., τ), the question remains as to what type they are. For instance, if a couple of concrete relations (e.g., β and δ) are collapsed into a single composite relation then that resulting composite relation must “express” (capture) the combined semantics of the ones it abstracts from. Given that UML supports a large set of relations (e.g., dependencies, associations, or aggregations), different combinations of them may yield different abstractions. This is why we see relation abstraction as semantically richer than classifier abstraction where, for the most part, only one type of classifier exists (e.g., classes for class diagrams, objects for object diagrams, packages for package diagrams). Abstracting classifiers of the same type usually yield a

classifier of that same type (e.g., two classes collapsed yield a composite but abstract class).

3. Semantic Abstraction

The main challenge during abstraction is to hide less important model elements and to only depict the remaining classifiers and their relations as part of the higher-level. The challenge we need to address is that on a concrete level, the dependencies between abstract model elements are not explicitly stated (e.g., as the dependencies between A and D where not explicitly stated in the concrete model (bottom) of Figure 1). Instead those dependencies are hidden within the lower-level model elements that we would like to hide. Relation abstraction, therefore, utilize a technique that allows groups of model elements (classifiers or relations) to be collapsed into high-level composite model elements that summarize their lower-level semantic dependencies. This paper will describe the patterns and rules necessary to do this.

3.1. Abstraction Examples

Take, for instance, Figure 2 which depicts two simple class diagrams. The first diagram (top) describes the relationships between *Compact Car*, *Car*, and *Driver*. It asserts that a *Car* is *operated-by* a *Driver* and that a *Compact Car* is a type of *Car*. Assuming that we do not care about the class *Car* but instead would like to know the direct relationship between *Compact Car* and *Driver* then we are actually asking for an abstracted version of that class diagram where the “helper class” *Car* and its relationships have been replaced by a simpler relationship. To find out whether there is indeed such a simpler relationship between *Compact Car* and *Driver*, we need to analyze the semantic dependencies between *Compact Car*, *Car*, and *Driver*. The information that a *Car* is operated by a *Driver* (association relationship) implies a property of the class *Car* (class properties are methods, attributes, and their relationships). The information that *Compact Car* is-a *Car* (inheritance) implies that *Compact Car* inherits all properties from *Car*. It follows that *Compact Car* inherits the association to *Driver* from *Car* which implies that a *Compact Car* is also operated by a *Driver*. This knowledge, of the transitive relationship between *Compact Car* and *Driver*, implies that the classifier *Car* as well as the relations *is-a* and *operated-by* could be collapsed into a composite, more abstract relationship linking *Compact Car* and *Driver* directly.

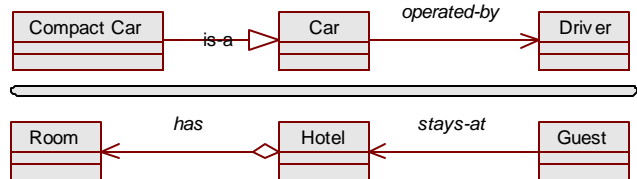


Figure 2. Class patterns

That composite relation should be of type *association*. This example shows a case where knowledge about the semantic properties of classifiers and relations allows us to eliminate a helper class and derive a more abstract class diagram. Above example therefore indicates a class abstraction pattern of the form:

```
Given: Inheritance-> Class -> Association
Implies: Association
```

This pattern may be used to collapse any occurrence of the “given” pattern into an occurrence of the “implies” pattern.

The second diagram in Figure 2 (middle) depicts a *Guest* who *stays-at* a *Hotel* which has *Rooms*. What the diagram does not depict is the (more abstract) relationship between *Guest* and *Rooms*. Semantically this diagram implies that *Rooms* are part of a *Hotel* which, in turn, implies that the class *Room* is conceptually *within* the class *Hotel*. If, therefore, *Guest* depends on *Hotel*, *Guest* also depends on *all* parts of *Hotel* – including *Rooms*. It follows that *Guest* relates to *Room* in the same manner as *Guest* relates to *Hotel*. We again found an abstraction pattern by analyzing the semantic relationships between the *Hotel*, *Guest*, and *Room* configuration:

```
Given: Association-> Class <- Aggregation
Implies: Association
```

Note that the directions of arrows have relevant semantic meanings. If *Hotel* would be part of *Room* then we could not automatically assume the correctness of above pattern.

3.2. Abstraction Patterns

For a transformation tool to automatically support abstraction, abstraction rules must be provided. Abstraction rules follow a simple structure. They define an input and an output pattern analogous to the given and implies pattern we used previously. Furthermore, the output pattern must be simpler (more abstract) than the input pattern, thus, guaranteeing that each applied abstraction rule indeed yields an abstraction.

A relation abstraction rule must have an input pattern of at least two relations with a classifier between them and

- | | |
|---|-----|
| (1) Generalization x Class x Generalization equals Generalizat. | 100 |
| (2) Generalization x Class x Dependency equals Dependency | 100 |
| (3) Generalization x Class x Association equals Association | 100 |
| (4) Generalization x Class x Aggregation equals Aggregation | 100 |
| (5) Dependency x Class x Generalization equals Dependency | 50 |
| (6) Dependency x Class x Dependency equals Dependency | 100 |
| (7) Dependency x Class x Association equals Association | 50 |
| (8) Association x Class x Association equals Association | 100 |
| (9) Association x Class x Aggregation equals Association | 100 |
| (10) Aggregation x Class x Generalization equals Aggregation | 50 |
| (11) Aggregation x Class x Dependency equals Dependency | 50 |
| (12) Aggregation x Class x Association equals Association | 90 |
| (13) Aggregation x Class x Aggregation equals Aggregation | 100 |
| (14) Dependency x Class x GeneralizationRev equals Dependency | 100 |
| (15) Dependency x Class x AggregationReverse equals Dependency | 80 |
| (16) GeneralizationRev x Class x Dependency equals Dependency | 50 |
| (17) GeneralizationRev x Class x Association equals Association | 70 |

Figure 3: Abstraction Rules for Class/Object Diagrams

an output pattern of at least a single relationship. Figure 2 gave two examples for relation abstraction (*Guest, Hotel, Room* and *Driver, Car, Compact Car*). Both input and output patterns are allowed to be more complex as long as the output pattern is simpler than the input pattern. If not, the abstraction algorithm could be non-deterministic.

3.3. Rules and Reliabilities

Figure 3 shows a list of input and output patterns (rules) for class abstraction. The left side depicts the class input patterns and the right side (after “equals”) depicts the class output patterns. Rule 3 in Figure 3 corresponds to the *Compact Car, Car, Driver* pattern and rule 17 corresponds to the *Guest, Hotel, Room* pattern. We also analyzed the semantic dependencies between other classes and their relationships and, thus, were able to derive 47 additional abstraction rules (not all of them could be listed in this paper). Note that the direction of relations is indicated through their name. If the relation name is used with no add-on, then a forward relation (a relation from left to right) is meant. If the string “Reverse” is added then a backward relation (a relation from right to left) is meant.

The number at the end of the rule indicates its reliability. Since patterns and rules are based on semantics, the 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. The reliability numbers can be between 0 and 100 (100 for high and 0 for low). Since composite model elements are derived through class abstractions rules and since those rules have reliability numbers attached, it follows that composite model elements inherit the reliability numbers from the rule(s) they were created from. For instance, if a composite relation was created through rule 17 then the composite relation has the reliability of 70.

3.4. Serial Abstractions

Serial abstractions were already implied previously when we talked about the possibility of applying multiple abstraction rules in sequence. Figure 1 illustrated such a case. There, on the top, a composite relation ω was created by abstracting another composite relation ζ as well as other relations. To do this, abstraction rules must be applied in sequence where the first abstraction yields an intermediate model and the second abstraction yields the final model. Serial abstraction therefore enables more complex abstraction tasks where a larger number of classes can be eliminated. Generally, all abstraction paths need to be explored. In case, different abstractions are found, all of them should be adopted.

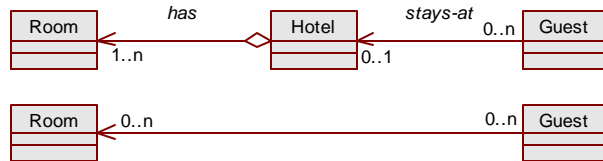


Figure 4. Cardinality Examples between Classes

3.5. Specialized Abstraction

Our generic abstraction mechanism works for box-and-arrow types of diagrams. Abstracting only boxes and arrows, however, only covers some features that diagrams incorporate. Figure 4, depicts a class diagram showing the relationships between *Hotel*, *Guest*, and *Room* again. Additionally, the figure depicts the cardinality between those classes. For example, a *Guest* may stay at one or many *Hotels* and a *Hotel* may have zero to many *Guests*. Also, a *Hotel* may have one to many *Rooms* and each *Room* belongs to one *Hotel* (the diamond at the end of that line shows a part-of relationship that has cardinality one unless defined otherwise). Cardinality issues are specific to class diagrams in UML. Its abstraction requires additional measures that do not apply to package and object diagrams. Cardinalities are abstracted by multiplying the left hand sides and right hand sides of relations separately. The multiplication itself is between the respective lower and upper-bounds. In our example, the left-hand-side cardinalities are “1..n” between *Hotel* and *Room* and “0..1” between *Guest* and *Hotel*. They must be multiplied (lower and upper ranges separately) to yield the abstract left hand side cardinality from *Guest* to *Room* “(0 * 1)..(1*n)” = “0..n.” Multiplying the right hand side is similar and yields “0..n.”

4. Discussion

Our relation abstraction method was implemented in a tool called UML/Analyzer (in the context of object and class diagrams) and was validated through a number of dimensions. First, we validated its rules by analyzing their semantic implications along the lines discussed in Section 3. Some of the models we used were provided by industry (e.g., the validation of a small part of a Satellite Telemetry Processing, Tracking, and Commanding System (TT&C) [5]). Other models were self-made. For instance, we reverse engineered and abstracted our own tools like AAA [6] and UML/Analyzer. We invented the relation abstraction technique in collaboration with Rational Software in 1997 [4]. Independently, however later, the group of Racz and Koskimies [7] came up with a class compression method that exploits the same concepts as our relation abstraction technique. They, however, did not create automated abstraction rules nor did they create tool support for automated abstraction in their work. Nevertheless, we see their work as another validation of ours since they independently made similar observations.

5. Conclusion

This paper introduced and discussed a class abstraction technique that is rule supported and automated. To date we demonstrated the usefulness of our technique in the context of class diagrams. Although our approach is based on syntactic patterns (boxes and arrows), the semantic meanings behind those patterns allow reasoning that makes our approach (and its rules) semantically rich.

We have also provided tool support to enable automated class abstraction. As input, our tool requires a concrete UML class or object diagram as well as a specification of what classes or objects are needed to be abstracted. In case of reverse engineering, both inputs must be provided by a human user. In case of consistency checking, this information can also be provided automatically (see [2]).

Acknowledgements

We wish to acknowledge Barry Boehm, Philippe Kruchten, and Nenad Medvidovic for helpful insights and discussions. This research is sponsored by DARPA through Rome Laboratory under contract F30602-94-C-0195 and by the Affiliates of USC-CSE.

References

1. Egyed, A. and Medvidovic, N.: "A Formal Approach to Heterogeneous Software Modeling," *Proceedings of 3rd Foundational Aspects of Software Engineering (FASE)*, March 2000.
2. Egyed, A.: "Heterogeneous View Integration and its Automation," PhD Dissertation, University of Southern California, Los Angeles, CA, May 2000.
3. Booch, G., Rumbaugh, J., Jacobson, I.: *The Unified Modeling Language User Guide*. Addison Wesley, 1999.
4. Egyed, A. and Kruchten, P.: "Rose/Architect: a tool to visualize architecture," , January 1999.
5. Alvarado, S.: "An Evaluation of Object Oriented Architecture Models for Satellite Ground Systems," *Proceedings of the 2nd Ground Systems Architecture Workshop (GSAW)*, February 1998.
6. Gacek, C.: "*Detecting Architectural Mismatches During System Composition*," PhD Dissertation, Center for Software Engineering, University of Southern California, Los Angeles, CA 90089-0781, USA, 1998.
7. Racz, F. D. and Koskimies, K.: "Tool-Supported Compression of UML Class Diagrams," *Proceedings of the 2nd International Conference on the Unified Modeling Language (UML)*, Fort Collins, CO, October 1999.