Automatically Discovering Transitive Relationships in Class Diagrams

Alexander Egyed

Teknowledge Corporation aegyed@ieee.org

Large-scale class diagrams are overwhelming to designers of software systems. They expose the designer to a level of detail that is often inappropriate for basic understanding ; and they complicate evolutionary changes in that the broader impact of changes is obscured by details. This chapter presents an approach for the automated abstraction of class diagrams that allows designers to 'zoom out' on class diagrams to investigate and reason about their bigger picture. The approach is based on a large number of abstraction rules that individually are not very powerful but, when used together, abstract complex class diagrams quickly. The technique was validated on over a dozen models where it was shown to be well-suited for model understanding, consistency checking, and reverse engineering.

1 INTRODUCTION

Refinement is often considered the natural course of software development where a problem is evolved into a solution. Yet, the more a class diagram is refined, the more there is a need to step back to investigate the bigger picture. We define abstraction to be the reverse of refinement. Abstraction is a transformation process [Met05] that transforms lower-level class diagrams into higher-level ones, containing fewer elements. Class abstraction has a number of vital uses. It allows designers to (1) focus a class diagram on a particular problem or goal, omitting details that are not needed in that context; and it allows designers to (2) zoom out on a class diagrams to investigate its entirety through a selected set of key elements.

In essence, abstraction is the simplification of models by removing details that are deemed unimportant by the designer. This naturally improves the understanding of class diagrams ; supports reverse engineering by transforming low-level models into higher-level ones; and supports consistency checking by comparing to existing higher-level models or architectures [EW01a] with abstracted ones. The technique may also help the designer in restructuring class diagrams [GZL05].

This chapter presents a technique for abstracting class diagrams [BRJ99] where designers decide which classes to keep (i.e., called *important classes*) and which ones to temporarily remove (called *helper classes*). Since it is not semantically correct to simply remove helper classes, the technique re-interprets the helper classes in terms of their effect on the important classes. A designer may guide the abstraction to emphasize different goals or concerns [Ham05].

Our technique computes how the important classes would interact with one another if their interaction were not obscured by the helper classes. The technique first identifies the paths of helper classes that span between any two important classes. These paths are then abstracted and replaced by single relationships that approximate the meanings of the paths. The technique is supported through small yet numerous abstraction rules that define how simple class/relationship paths are replaced by single relationships. The application of an abstraction rules is guided by the paths of helper classes that span between any two important classes. These paths are abstracted individually by applying the abstraction rules in the order in which classes are traversed while one important class interacts with another (through these helper classes). This problem is similar to the graph transformation discussed in [GGZ+05] but avoids the pattern matching problem by dealing with strings of classes only.

We evaluated the technique on over a dozen real-world case studies ranging from in-house developed models to third-party models. Most notably, we used the technique in connection with the Inter-Library Loan System [AAHK99], a part of a Satellite Ground System [Alv98], C2SADEL to UML integration [EW01a], Video-On-Demand System, SDS Statechart Simulator [EW01b], and other projects. The sizes of the models ranged from several dozen to several hundred classes. The validation showed that the technique produces correct abstractions 96% of the time. This chapter presents the technique, originally introduced in [Egy02], and then discusses two key extensions:

- The number of paths between important classes rises exponentially with the number of helper classes involved. We present an optimization that identifies in linear time the actual classes and relationships used that reach between any two helper classes; thus minimizing the path exploration problem.
- Complete paths are no longer computed before abstraction (i.e., due to the exponential problem) but are stepwise abstracted while the paths are explored. This avoids unnecessary path exploration because a partially non-abstractable path is also not abstractable in its entirety.

Also, a new tool was built as an add-in to IBM Rational Rose™. Rose is used to draw class structures and, using Rose's selection mechanism, a designer selects important classes for abstraction. The abstraction results are visualized in Rose also.

2 ILLUSTRATIVE EXAMPLE

For illustrative purposes, this chapter uses a simple UML class diagram [BRJ99] of a Hotel Management System (HMS) that provides support for reservations, check-in/check-out procedures, and associated financial transactions (see Figure 1). The figure defines that a *Person* may have an *Account* and that a single account may belong to multiple persons; it also defines that an account may have *Transactions* and transactions may be either *Expenses* or *Payments*. Furthermore, it defines that a *Guest* is a *Person* who provides services for reservation and check in/out procedures. Both *Room* and *Reservation* are part of *Hotel* to indicate that instances of *Room* and *Reservation* are unambiguously associated with particular instances of *Hotel*. *Guest* is also related to *Room* and *Reservation* but less tightly via calling dependencies. These two calling dependencies describe that an instance of *Guest* may stay at a *Room* of a *Hotel* or may have several *Reservations* for any given *Hotel*.

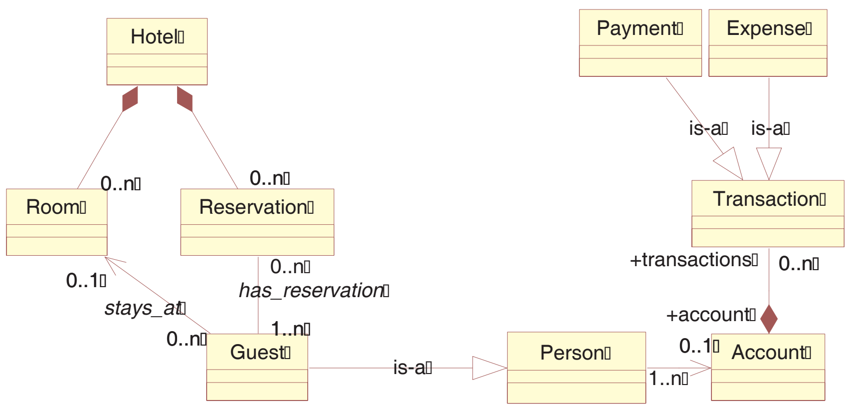

Fig. 1. Illustrative Class Diagram of a Hotel Management System (HMS)

While this class diagram is simple enough for human comprehension , we have worked with class diagrams that include thousands of classes and many more relationships. It is impossible for humans to comprehend such class structures and designers resort to abstraction as a means of coping with this complexity. Abstraction allows a designer to depict class structures from a particular point of view, concern, requirement, or other form of interest.

Figure 2 depicts a couple abstractions of Figure 1 that emphasize on different sets of important classes. For example, Figure 2 (a) depicts the important classes *Guest*, *Payment*, and *Expense* but it also depicts relationships among these three classes that are not to be found in Figure 1. These relationships are the abstract interpretation of the hidden classes. Figure 2 (b) and (c) depict yet other abstractions that 'slice' across the classes in Figure 1.

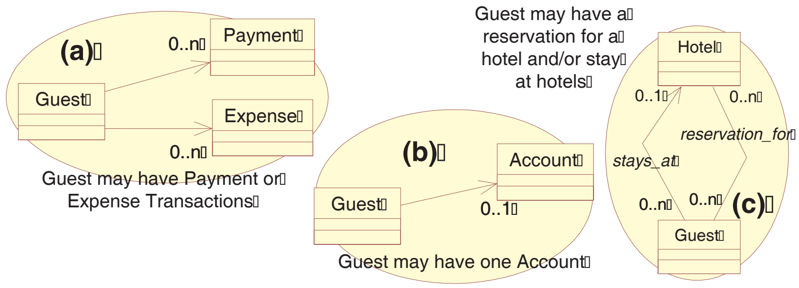

Fig. 2. Abstractions of the HMS system

Clearly, there are a range of benefits associated with working with abstractions. Each abstraction depicts a 'slice' of the class diagram and is easier to understand. Also, in this case, each abstraction relates to some form of requirement or system goal and designers may intuitively benefit from seeing the HMS in terms of these individual goals.

3 SIMPLE ABSTRACTION

The main goal of class abstraction is to hide information from a class diagram that is perceived as not important. Since designers likely have different notions as to what classes are important (i.e., reflecting different goals or problem), a class abstraction technique needs to be guidable. Guidance may be as simple as a designer selecting model elements that are of particular interest; or guidance may be provided via trace dependencies [Egy03][GF94]. In the following, we presume that such guidance is available.

Abstraction replaces all unimportant helper classes in the class diagram such that the resulting diagram depicts only the important classes and their computed relationships. The main challenge of abstraction is to compute relationships out of the helper classes. That is, if we simply hide the class *Reservation* and its relationships in Figure 1 then the class diagram loses the knowledge that a *Guest* may have a *reservation for* a *Hotel*.

This section presents generic abstraction rules (patterns) that are based on the UML notation for class diagrams [BRJ99]. Currently supported are class diagrams with four types of relationships: generalization (inheritance), association (calling direction), aggregation (part-of), and dependency. The presented abstraction rules are generic and applicable to a wide range of software projects. Designers are not required to extend or modify this rule set unless they wish to fine tune it (e.g., domain specific rules).

3.1 Semantic Rules

The class abstraction technique interprets the transitive meaning of classes and their relationships. For example, the information that a *Person* may have an *Account* (association relationship in Figure 1) implies a property of the class *Person* (class properties are methods, attributes, or relationships). Furthermore, the information that *Guest* is-a *Person* (inheritance) implies that *Guest* inherits all properties from *Person*. It follows that *Guest* inherits the association to *Account* from *Person* implying that a *Guest* may have zero or one *Account*s. This knowledge of the transitive relationship between *Guest* and *Account* (via *Person*) implies that the class *Person* (and its two relationships to *Guest* and *Account*) could be 'collapsed' into a composite, more abstract relationship linking *Guest* and *Account* directly. That composite relationship should be of type 'association' with the cardinality '0..1'. This example shows a case where knowledge about the semantic properties of classes and relationships makes it possible to eliminate a class and derive a slightly more abstract class diagram. The example above can be seen as a class abstraction pattern of the following form (cardinalities are discussed later):
{1} GeneralizationRight - Class - AssociationRight -> AssociationRight

We use relationship names post-fixed with either 'Left' or 'Right' to indicate directionality. 'GeneralizationRight - Class AssociationRight' implies a generalization relationship terminating in the given class and an association relationship originating from that same class. On the other hand, 'GeneralizationRight - Class AssociationLeft' implies both generalization and association relationships terminating in the same class. Given that the above abstraction rule captures an observation that is universally true (meaning true for all instances), this rule collapses any occurrence of the given pattern (before ->) into an occurrence of the implies pattern (after ->).

The transitive property of inheritance may also be used for other types of relationships. For instance, *Guest* could also inherit other relationships from *Person* (e.g., aggregation, dependency, or reverse association relationships - see rules 2-7 below).
{2} GeneralizationRight - Class - DependencyRight -> DependencyRight
{3} GeneralizationRight - Class - AssociationRight -> AssociationRight
{4} GeneralizationRight - Class - [Agg]Assoc.Right -> [Agg]Assoc.Right
{5} GeneralizationRight - Class - DependencyLeft -> DependencyLeft
{6} GeneralizationRight - Class - AssociationLeft -> AssociationLeft
{7} GeneralizationRight - Class - [Agg]Assoc.Left -> [Agg]Assoc.Left
{8} GeneralizationRight - Class - Association -> Association

UML class relationships are usually uni-directional requiring us to differentiate 'Left' from 'Right'. The only exception is the association relationship which may also be bi-directional. Rule 8 in the above block of patterns states that the bi-directionality of the association is maintained if abstracted together with a generalization.
{9} GeneralizationRight - Class - GeneralizationRight -> GeneralizationRight

The previous assumption about inheritance is true for all relationship types except for generalization relationships. On the one hand, it is valid to state that A inherits from C if A inherits from B and B inherits from C (see rule 9); however, if both A and C inherit from B (A and C share a common parent) then transitively this does not imply a relationship between A and C. It follows that no relationship exists between *Payment* and *Expense* in Figure 1. Similar restrictions apply if two classes share a common child (multiple inheritance). Rules 10 and 11 express these situations. The symbol 'Ø' is used to indicate that no abstraction is possible.
{10} GeneralizationRight - Class - GeneralizationLeft -> Ø
{11} GeneralizationLeft - Class - GeneralizationRight -> Ø

To find more abstraction rules, consider the relationship between *Guest* and *Hotel* in Figure 1. The class diagram uses the class *Reservation* to define that a *Guest* may have *reservation for* a *Hotel*. If a designer were to derive the transitive relationship from *Guest* to *Hotel* through *Reservation* then the helper class *Reservation* and its relationships need to be replaced. In order to do that, it is again necessary to investigate the transitive meaning of the to-be-replaced model elements. The class diagram shows the class *Hotel* with an aggregation relationship from *Reservation* to *Hotel* implying that *Reservation* is a part of *Hotel*. The class diagram also defines that *Guest* has an association relationship to *Reservation* (instance of *Guest* may call instance of *Reservation*). Given that *Reservation* is a part of *Hotel* implies that the class *Reservation* is conceptually within the class *Hotel*. If, therefore, *Guest* depends on *Reservation* and *Reservation* is part of *Hotel* then *Guest* must also depend on *Hotel*. It follows that *Guest* relates to *Hotel* in the same manner as *Guest* relates to *Reservation*. We thus have found another abstraction rule (rule 12). As before, the same reasoning is applied to other relationships (e.g., rules 13-15):
{12} Association - Class - Association [Agg] -> Association
{13} AssociationRight - Class - Association [Agg] -> AssociationRight
{14} AssociationLeft - Class - AssociationLeft[Agg] -> AssociationLeft
{15} AssociationLeft[Agg] - Class Assoc.Left[Agg] -> Assoc.Left[Agg]

Note that aggregations are UML associations with the aggregate property [Agg] at one of its ends. The directionality of aggregations also has relevant semantic meaning. For example, if *Hotel* were part of *Reservation* then one could not readily apply the above patterns (e.g., as with the relationship between *Person* and *Transaction* in Figure 1).

3.2 Living with Ambiguous Class Definitions

The example of determining the relationship between *Person* and *Transaction* (Figure 1) introduces a new challenge. If one were to derive the transitive relationship between *Person* and *Transaction* then one would need to abstract away the helper class *Account* and its relationships. *Person* currently has an association to *Account* and *Transaction* is part of *Account* ('AssociationRight

- Class - [Agg]Association'). By *Person* having an association to *Account* one could argue that *Person* relates to every part of *Account*. Since *Transaction* is a part of *Account* it follows that *Person* must also relate to *Transaction*. Although this argument is true in many situations, it is flawed nonetheless. We make the assumption that by *Person* relating to *Account* it relates to all its parts. It is however conceivable that *Person* relates to a subset of *Account* only - a subset other than *Transaction* (i.e., mostly the case where classes provide independent services, e.g., a math library).

Taking a more critical stance towards our abstraction rules, one may find that this is not the first case of uncertainty. Consider again the very first rule 1 'GeneralizationRight - Class - AssociationRight -> AssociationRight'. Previously, it was stated that *Guest* has an association relationship to *Account* simply because it inherited one from *Person*. To illustrate this reasoning more precisely, assume that *Person* has a method 'foo' that creates an instance of *Account* ('0..1' association between *Person* and *Account*). Based on that assumption, surely, one can infer that *Guest* also has a '0..1' association relationship to *Account* because *Guest* inherits method 'foo' from *Person*. Yet the flaw in this reasoning becomes apparent if *Guest* inherits method 'foo' but overwrites its body such that it no longer creates an instance of class *Account* nor calls the overwritten method of the parent class. In such a case, *Guest* would not inherit the '0..1' association relationship from *Person* to *Account*. Abstracting the pattern 'GeneralizationRight - Class AssociationRight' is thus 'AssociationRight' in some cases but not abstractable (no relationship) in other cases.

Observations such as this one naturally cause a dilemma. We are opposed to using abstraction rules that are not 100% reliable but we encounter imprecise model definitions that take away from our ability to reason precisely. We refer to these uncertainties as 'model ambiguities' because imprecise model definitions lead to potentially different, ergo ambiguous interpretations. A simple solution to this ambiguity problem is to create a semi-automated abstraction process that lets the designer decide in case of uncertainty (e.g., [RK99]). Given the large and complex nature of models, semi-automated abstraction becomes very costly. Indeed, it has been our observation that not computing time but human-intervention constitutes key complexities in activities such as model transformation and consistency checking. A similar unsatisfactory solution to this problem is to make arbitrary decisions about the most likely abstraction case and ignore less likely scenarios (e.g., ignore that the child may overwrite method 'foo' of the parent). This solution is unsatisfactory because it makes our approach less reliable producing potentially erroneous abstraction results without the designer being aware of it.

UML class diagrams, like many other graphical description languages, are somewhat imprecise and ambiguous [JR00]. Indeed, we find that their relaxed nature often encourages their use since designers are sometimes either unable or unwilling to make precise design decisions. For instance, in UML it cannot be modeled whether class A overwrites methods it inherits from class B.

Although a lack of precision on part of UML, one may argue that it may not always be obvious during design time when to overwrite methods. More recent research has shown that formal annotations can improve the precision of UML (or alike notations) [EFLR98][Ö98][MC01] but their use is generally optional and left to the discretion of the designer.

Since the basic notation of UML is ambiguous, we take the stance that automated abstraction needs to handle ambiguities. Our solution to the ambiguity problem is to maintain the ambiguity during abstraction. For instance, if it is unknown whether methods get overwritten during inheritance then we argue that 'GeneralizationRight - Class AssociationRight' is 'AssociationRight' in some cases and 'Ø' (no relationship) in other cases. This implies that abstract relationships indicated by our approach may or may not factually exist. In cases where more complex abstractions allow multiple abstract interpretations, our approach suggests all of them to indicate this uncertainty (ambiguity). Our solution has the advantage that no abstract results are omitted although false positives may happen. Section 6 will show that the likelihood of false positives is very low (4%). Note that an alternative solution would be to use a subset of abstraction rules that are known to be 100% correct. The problem with this alternative solution is that only very few such rules exist and large-scale abstraction would be rather ineffective as a consequence.

3.3 Other Abstraction Rules

Thus far we focused on class patterns that use generalization and aggregation relationships. In the following, we briefly discuss some abstraction patterns that use association and dependency relationships (please refer to [Egy03] for more details).

An association relationship describes calling operations among classes. For instance, *Person* having an association relationship to *Account* implies that a method of *Person* may call methods of *Account*. If class A calls methods of class B and class B calls methods of class C then, transitively, class A might also call methods of class C ('AssociationRight - Class - AssociationRight -> AssociationRight'). In case an uni-directional association is abstracted together with a bi-directional association, the bi-directionality is replaced. For instance, if class A can only call class B but classes B and C can call one another then, transitively, class A can still only call class C but not the other way around ('AssociationRight - Class - Association -> AssociationRight').

Dependency relationships are used in UML to indicate a required presence of classes. For instance, if class A depends on class B then class B must be present for class A to function. The notion of a dependency is other than calling (association) and is used, i.e., to single out classes that are used as parameters in method calls (i.e., class A does not call class B but class A has a method that expects an instance of class B as a parameter). It is thus safe to state that 'DependencyRight - Class DependencyRight' must also abstract

Table 1. Complete List of Abstraction Rules for Class Diagrams

1. GeneralizationRight - Class - GeneralizationRight -> GeneralizationRight
2. GeneralizationRight - Class - DependencyRight -> DependencyRight
3. GeneralizationRight - Class - AssociationRight -> AssociationRight
4. GeneralizationRight - Class - [Agg]AssociationRight -> [Agg]AssociationRight
5. GeneralizationRight - Class - GeneralizationLeft -> Ø
6. GeneralizationRight - Class - DependencyLeft -> DependencyLeft
7. GeneralizationRight - Class - AssociationLeft -> AssociationLeft
8. GeneralizationRight - Class - AssociationLeft[Agg] -> AssociationLeft[Agg]
9. GeneralizationRight - Class - Association -> Association
10. GeneralizationRight - Class - [Agg]Association -> [Agg]Association
11. GeneralizationRight - Class - Association[Agg] -> Association[Agg]
12. GeneralizationLeft - Class - GeneralizationRight - Class-> Ø
13. GeneralizationLeft - Class - DependencyRight -> Ø
14. GeneralizationLeft - Class - AssociationRight -> Ø
15. GeneralizationLeft - Class - [Agg]AssociationRight -> Ø
16. GeneralizationLeft - Class - GeneralizationLeft -> GeneralizationLeft
17. GeneralizationLeft - Class - DependencyLeft -> DependencyLeft
18. GeneralizationLeft - Class - AssociationLeft -> AssociationLeft
19. GeneralizationLeft - Class - AssociationLeft[Agg] -> AssociationLeft[Agg]
20. GeneralizationLeft - Class - Association -> AssociationLeft
21. GeneralizationLeft - Class - [Agg]Association -> AssociationLeft
22. GeneralizationLeft - Class - Association[Agg] -> AssociationLeft[Agg]
23. DependencyRight - Class - GeneralizationRight -> DependencyRight
24. DependencyRight - Class - DependencyRight -> DependencyRight
25. DependencyRight - Class - AssociationRight -> DependencyRight
26. DependencyRight - Class - [Agg]AssociationRight -> DependencyRight
27. DependencyRight - Class - GeneralizationLeft -> DependencyRight
28. DependencyRight - Class - DependencyLeft -> Ø
29. DependencyRight - Class - AssociationLeft -> Ø
30. DependencyRight - Class - AssociationLeft[Agg] -> Ø
31. DependencyRight - Class - Association -> DependencyRight
32. DependencyRight - Class - [Agg]Association -> DependencyRight
33. DependencyRight - Class - Association[Agg] -> DependencyRight
34. DependencyLeft - Class - GeneralizationRight -> Ø
35. DependencyLeft - Class - DependencyRight -> Ø
36. DependencyLeft - Class - AssociationRight -> Ø
37. DependencyLeft - Class - [Agg]AssociationRight -> Ø
38. DependencyLeft - Class - GeneralizationLeft -> DependencyLeft
39. DependencyLeft - Class - DependencyLeft -> DependencyLeft
40. DependencyLeft - Class - AssociationLeft -> DependencyLeft
41. DependencyLeft - Class - AssociationLeft[Agg] -> DependencyLeft
42. DependencyLeft - Class - Association -> DependencyLeft
43. DependencyLeft - Class - [Agg]Association -> DependencyLeft
44. DependencyLeft - Class - Association[Agg] -> DependencyLeft
45. AssociationRight - Class - GeneralizationRight -> AssociationRight
46. AssociationRight - Class - DependencyRight -> DependencyRight

47. AssociationRight - Class - AssociationRight -> AssociationRight
48. AssociationRight - Class - [Agg]AssociationRight -> AssociationRight
49. AssociationRight - Class - GeneralizationLeft -> AssociationRight
50. AssociationRight - Class - DependencyLeft -> Ø
51. AssociationRight - Class - AssociationLeft -> Ø
52. AssociationRight - Class - AssociationLeft[Agg] -> Ø
53. AssociationRight - Class - Association -> AssociationRight
54. AssociationRight - Class - [Agg]Association -> AssociationRight
55. AssociationRight - Class - Association[Agg] -> AssociationRight
56. AssociationLeft - Class - GeneralizationRight -> Ø
57. AssociationLeft - Class - DependencyRight -> Ø
58. AssociationLeft - Class - AssociationRight -> Ø
59. AssociationLeft - Class - [Agg]AssociationRight -> Ø
60. AssociationLeft - Class - GeneralizationLeft -> AssociationLeft
61. AssociationLeft - Class - DependencyLeft -> DependencyLeft
62. AssociationLeft - Class - AssociationLeft -> AssociationLeft
63. AssociationLeft - Class - AssociationLeft[Agg] -> AssociationLeft
64. AssociationLeft - Class - Association -> AssociationLeft
65. AssociationLeft - Class - [Agg]Association -> AssociationLeft
66. AssociationLeft - Class - Association[Agg] -> AssociationLeft
67. [Agg]AssociationRight - Class - GeneralizationRight -> [Agg]AssociationRight
68. [Agg]AssociationRight - Class - DependencyRight -> DependencyRight
69. [Agg]AssociationRight - Class - AssociationRight -> AssociationRight
70. [Agg]AssociationRight - Class - [Agg]AssociationRight -> [Agg]AssociatRight
71. [Agg]AssociationRight - Class - GeneralizationLeft -> [Agg]AssociationRight
72. [Agg]AssociationRight - Class - DependencyLeft -> Ø
73. [Agg]AssociationRight - Class - AssociationLeft -> Ø
74. [Agg]AssociationRight - Class - AssociationLeft[Agg] -> Ø
75. [Agg]AssociationRight - Class - Association -> AssociationRight
76. [Agg]AssociationRight - Class - [Agg]Association -> [Agg]AssociationRight
77. [Agg]AssociationRight - Class - Association[Agg] -> AssociationRight
78. AssociationLeft[Agg] - Class - GeneralizationRight -> Ø
79. AssociationLeft[Agg] - Class - DependencyRight -> Ø
80. AssociationLeft[Agg] - Class - AssociationRight -> Ø
81. AssociationLeft[Agg] - Class - [Agg]AssociationRight -> Ø
82. AssociationLeft[Agg] - Class - GeneralizationLeft -> AssociationLeft[Agg]
83. AssociationLeft[Agg] - Class - DependencyLeft -> DependencyLeft
84. AssociationLeft[Agg] - Class - AssociationLeft -> AssociationLeft
85. AssociationLeft[Agg] - Class - AssociationLeft[Agg] -> AssociationLeft[Agg]
86. AssociationLeft[Agg] - Class - Association -> AssociationLeft
87. AssociationLeft[Agg] - Class - [Agg]Association -> AssociationLeft
88. AssociationLeft[Agg] - Class - Association[Agg] -> AssociationLeft[Agg]
89. [Agg]Association - Class - GeneralizationRight -> [Agg]AssociationRight
90. [Agg]Association - Class - DependencyRight -> DependencyRight
91. [Agg]Association - Class - AssociationRight -> AssociationRight
92. [Agg]Association - Class - [Agg]AssociationRight -> [Agg]AssociationRight
93. [Agg]Association - Class - GeneralizationLeft -> [Agg]Association

94. [Agg]Association - Class - DependencyLeft -> DependencyLeft
95. [Agg]Association - Class - AssociationLeft -> AssociationLeft
96. [Agg]Association - Class - AssociationLeft[Agg] -> AssociationLeft
97. [Agg]Association - Class - Association -> Association
98. [Agg]Association - Class - [Agg]Association -> [Agg]Association
99. [Agg]Association - Class - Association[Agg] -> Association

---

100. Association[Agg] - Class - GeneralizationRight -> AssociationRight
101. Association[Agg] - Class - DependencyRight -> DependencyRight
102. Association[Agg] - Class - AssociationRight -> AssociationRight
103. Association[Agg] - Class - [Agg]AssociationRight -> AssociationRight
104. Association[Agg] - Class - GeneralizationLeft -> Association[Agg]
105. Association[Agg] - Class - DependencyLeft -> DependencyLeft
106. Association[Agg] - Class - AssociationLeft -> AssociationLeft
107. Association[Agg] - Class - AssociationLeft[Agg] -> AssociationLeft[Agg]
108. Association[Agg] - Class - Association -> Association
109. Association[Agg] - Class - [Agg]Association -> Association
110. Association[Agg] - Class - Association[Agg] -> Association[Agg]

---

111. Association - Class - GeneralizationRight -> AssociationRight
112. Association - Class - DependencyRight -> DependencyRight
113. Association - Class - AssociationRight -> AssociationRight
114. Association - Class - [Agg]AssociationRight -> AssociationRight
115. Association - Class - GeneralizationLeft -> Association
116. Association - Class - DependencyLeft -> DependencyLeft
117. Association - Class - AssociationLeft -> AssociationLeft
118. Association - Class - AssociationLeft[Agg] -> AssociationLeft
119. Association - Class - Association -> Association
120. Association - Class - [Agg]Association -> Association
121. Association - Class - Association[Agg] -> Association

to a 'DependencyRight'. Since dependencies can also be inherited (generalization) and a dependency of a part also implies a dependency of the whole (aggregation) the usual assumptions can be made about their abstraction.

3.4 The Complete List

To date, we have validated our approach in context of UML class diagrams and the relationship types: generalization, association, dependency, and aggregation. Considering directionality, this implies eight uni-directional relationship types such as 'GeneralizationRight' or 'AggregationLeft' plus three bi-directional relationship types 'Association', '[Agg]Assocition', and 'Association[Agg]'. Altogether, those relationships may form 121 different patterns (11*11). Table 1 gives the complete list of abstraction patterns as they are currently defined in our approach.

It is interesting to observe that 29 patterns cannot be abstracted whereas the remaining 92 patterns have abstract counterparts. Given that it should not matter from what direction a pattern is viewed (or abstracted) it follows that mirror images of abstraction patterns must have the same values. For instance, the pattern 'GeneralizationRight - Class GeneralizationRight' (rule 1) is equivalent to the pattern 'GeneralizationLeft - Class GeneralizationLeft' (rule 16).

4 COMPOSITE ABSTRACTION

The previous section discussed abstraction in context of numerous simple rules. These abstraction rules are not very powerful but this section demonstrates how complex class diagrams are abstracted using those rules.

4.1 Path Abstraction

We refer to a sequence of helper classes between two important classes as a path. Abstraction rules are serialized to abstract a path of classes. Figure 3 (top) depicts a path from *Guest* to *Payment*; taken from Figure 1. If it is of interest to know the transitive relationship between *Guest* and *Payment* then the abstraction rules in Table 1 have to be applied in sequence to eliminate the helper classes *Person*, *Account*, and *Transaction*.

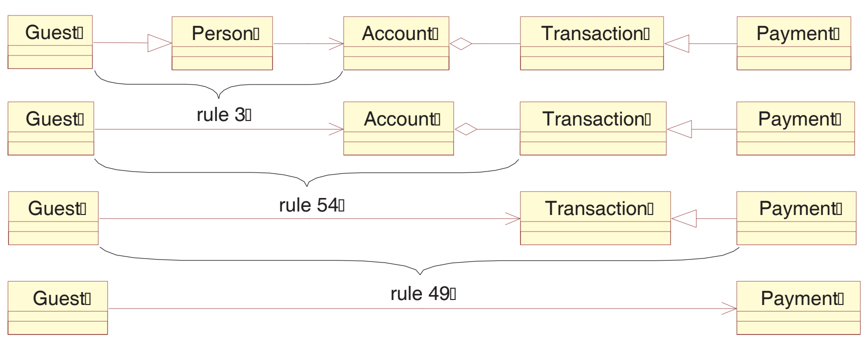

Fig. 3. Abstraction of a Path of Classes

The abstraction rules are applied in the order in which the classes are visited. That is, if *Guest* calls *Transaction* then this call first passes through *Person*, then through *Account*, next through *Transaction*, before, finally, reaching *Payment*. Thus, rule 3 (Table 1) is applied first to replace *Person* (and its relationships to *Guest* and *Account*); rule 54 is applied next to replace *Account*, and rule 49 is applied finally to replace *Transaction*. As a result, we discover a transitive association relationship between *Guest* and *Payment*. Of course, the path could also be explored in reverse from *Payment* to *Guest*.

The abstraction of a path is different from [Egy02] in that no longer all path combinations are explored. This was previously done to ensure that all rules are treated equally during abstraction. For example, the problem in Figure 3 could have also been resolved by applying the rules in a different order. That is, rule 54 could have been applied first (replacing *Account*), rule 3 next (replacing *Person*), and rule 49 finally (replacing *Transaction*). In this particular case, the outcome of the abstraction would have been identical. While there is no guarantee that all combinations of all rules produce identical abstraction results, we have not encountered a case where it would. Furthermore, given that it is computationally expensive to explore all combinations of rules, the technique now resolves a path in the order it is traversed.

4.2 Paths among Neighboring Important Classes

A path is a sequence of helper classes between two important classes. A path should not contain important classes. Consider, for example, that a designer is interested in the relationships among the important classes *Hotel*, *Guest*, *Payment*, and *Expense*; ignoring helper classes such as *Account*, *Reservation*, or *Person*. If one were to investigate all paths among *Hotel*, *Guest*, *Payment*, and *Expense* then one would find nine (Figure 4 (top)): one path between *Payment* and *Expense*, one path between *Payment* and *Guest*, one path between *Expense* and *Guest*, two paths between *Hotel* and *Guest*, two paths between *Hotel* and *Payment*, and two paths between *Hotel* and *Expense*.

All nine paths reflect ways for the important classes *Hotel*, *Guest*, *Payment*, and *Expense* to interact with one another. However, it is not desired to know about all possible transitive relationships among all classes of the *Hotel-Guest-Payment-Expense* set. Instead it is desired to know about transitive relationships between neighboring important classes only. Take for instance the path *Hotel-Reservation-Guest-Person-Account-Transaction-Expense* in Figure 4 (top). Abstracting this path reveals a uni-directional association from *Hotel* to *Expense*. However, this path also eliminates the important class *Guest* because it is part of this path between *Hotel* and *Expense*. This is invalid here since one should not declare *Guest* an important class for abstraction but at the same time eliminate it in some abstraction path. This is invalid because the abstract relationship between *Hotel* & *Expense* would be redundant with other abstract relationships between *Hotel* & *Guest* and *Guest* & *Expense* where the former relationship (*Hotel-Expense*) is an abstraction of the latter two relationships. That is, *Hotel* is not capable of calling *Expense* directly but requires *Guest*. It is thus sufficient to show that *Hotel* calls *Guest* and that *Guest* calls *Expense*. Figure 4 (bottom) depicts the subset of paths from Figure 4 (top) that do not contain any important classes.

This restriction of paths is also important for computational scalability. That is, the number of paths would increase quadratically with the number of important classes. Yet, if one is not interested in all transitive relationships among all important classes; but only the transitive relationships among

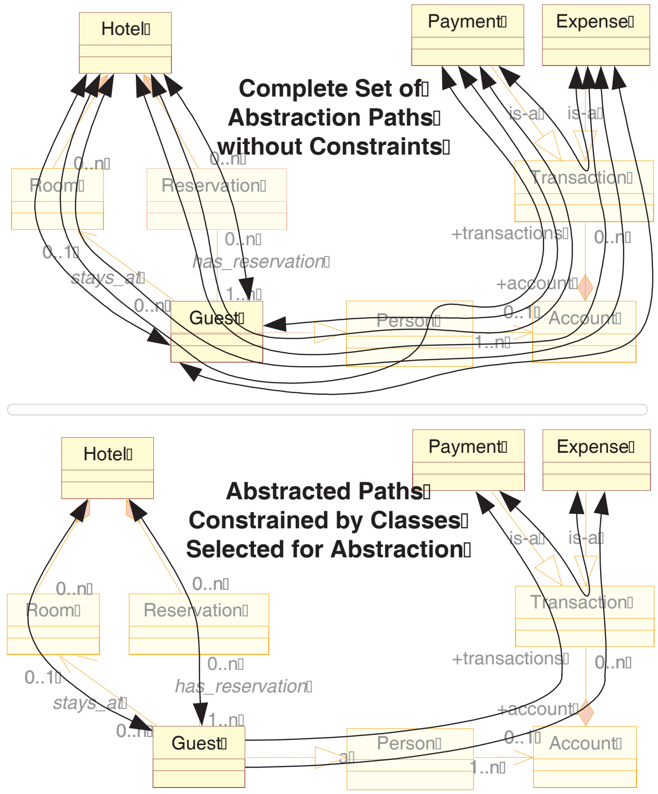

Fig. 4. All Transitive Relationships (top) and Transitive Relationships of Neighboring Classes (bottom)

neighboring important classes then this significantly reduces the number of paths (i.e., the number of neighboring classes is a relative constant that is not affected by the number of important classes). Usability is also increased because the number of abstraction results is reduced by the same degree.

4.3 Abstracting Cardinalities

The rules in Table 1 emphasize the syntactic nature of 'boxes' and 'arrows' in class diagrams. However, class diagrams consist of more than just boxes and arrows. Figure 5 (left), depicts the familiar class diagram of the HMS system showing the relationships among *Hotel*, *Guest*, *Reservation*, and *Room*. Additionally, the figure depicts the cardinalities among those classes as they were originally introduced in Figure 1. For example, it shows that a *Guest* may stay at zero or one *Rooms* and may have zero, one, or more *Reservations*. Also, a *Hotel* may have zero, one, or many *Rooms* and each *Room* belongs to exactly one *Hotel* (the diamond head of the aggregation relationship has cardinality one unless defined otherwise).

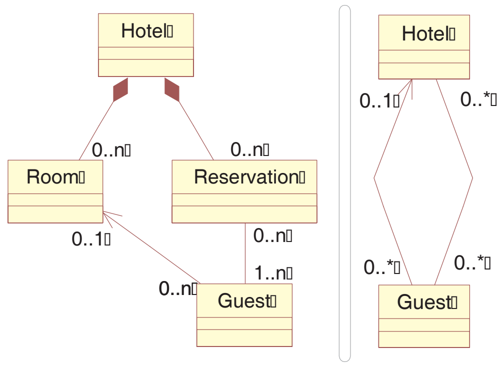

Fig. 5. Abstracting Cardinalities

Figure 5 (right) shows that the *Hotel*-*Room*-*Reservation*-*Guest* class structure is abstractable into two relationships between the important classes *Guest* and *Hotel*: a bi-directional association and a uni-directional association (Figure 5 (right)). These two relationships are based on the two distinct paths that exist between *Guest* and *Hotel*. Since an instance of *Guest* interacts with zero or one instances of *Room* and, in turn, an instance of *Room* is always associated with exactly one instance of *Hotel* (semantic implication of aggregation relationship) a *Guest* may stay at zero or one *Hotel*s at any given point in time. This is a transitive property. Since associations and aggregations have two ends, there are always exactly two cardinalities one has to consider. The second cardinality investigates the reverse where an instance of *Hotel* may interact with zero, one, or more instances of *Room* and an instance of *Room* may interact with zero, one, or more instances of *Guest*. This implies that multiple guests may stay at any given *Hotel* room. Other cardinality scenarios follow the same pattern and are described in more detail in [Egy02].

4.4 Path Exploration

Section 4.2 discussed how important classes restrict the exploration of paths in that only paths among neighboring important classes are considered. In spite of this restriction, path exploration is still a computationally expensive activity as there are often many paths between any two neighboring important classes. We address this problem in the form of two optimizations that avoid unnecessary path explorations.

The first optimization computes in advance what helper classes (and their relationships) are part of paths between two given important classes. For example, if it is desired to understand the transitive relationships among *Guest*,

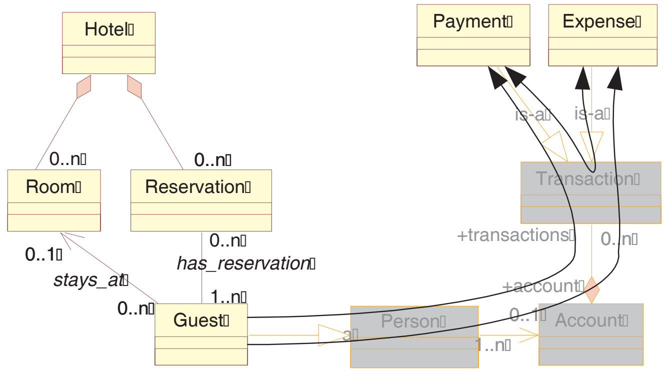

Fig. 6. Helper Classes and Paths between Important Classes

Payment, and *Expense* (Figure 6) then it is not necessary to explore paths that involve classes such as *Room* or *Hotel*. Yet, the path exploration algorithm in [Egy02] explored paths recursively without knowing, in advance, what paths would succeed in connecting the desired important classes. The technique now computes in linear time what classes bridge between desired important classes. In the case of the example, the technique computes that only the classes *Person*, *Account*, and *Transaction* bridge the important classes *Guest*, *Payment*, and *Expense*. All other helper classes are ignored during the path exploration.

The second optimization limits the exploration of paths by continuously evaluating the abstractability of paths. That is, there are potentially many paths between any two helper classes but most of them are not abstractable. It is thus not necessary to explore a path in its entirety if a partial exploration and abstraction of that path already determines that it is not abstractable (i.e., if a path is not abstractable partially then it is also not abstractable in its entirety). This optimization avoids the exploration of subsequent path alternatives not yet explored.

The two optimizations discussed in this section minimize the path exploration problem in that only those helper classes are considered that yield useful paths and paths are only abstracted for as long as they are abstractable.

5 AUTOMATION AND TOOL SUPPORT

The abstraction technique requires a designer to guide the abstraction by defining important classes in a class diagram. The technique then replaces the remaining helper classes with transitive relationships. The tool support fully automates the replacement of helper classes. This reduces manual effort and makes abstraction results reproducible. This section describes the tool we developed.

Our approach was co-developed with Rational Software [EK99] who developed a tool called Rose/Architect (construction of Rose/Architect was subcontracted to Ensemble Systems by Rational Corporation). Since then, the approach was extended and the author developed another non-proprietary tool. The new tool was integrated with IBM Rational Rose™for the purpose of creating and modifying diagrams. Designers mark important classes through Rose's selection mechanism. The abstraction results are then visualized in Rose. Figure 7 depicts two screen snapshots of the tool. The top of Figure 7 shows a class diagram of the HMS as defined in Rose. The bottom of Figure 7 shows Rose visualizing an abstracted class diagram with the important classes *Hotel*, *Guest*, *Payment*, and *Expense*. The abstraction tool was integrated with Rose using an integration framework developed at Teknowledge Corporation for integrating software components with COTS (commercial-off-the-shelf) components.

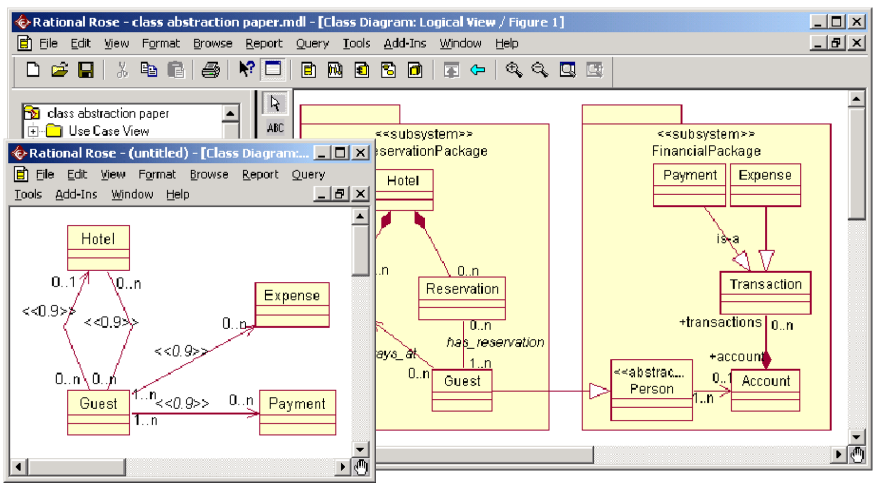

Fig. 7. Class Abstraction Tool integrated with IBM Rational Rose ™

6 VALIDATION

This section discusses the validity of our approach in terms of its correctness, and manual overhead.

6.1 Validity of Abstraction Rules and Algorithm

We evaluated the validity of abstraction results on a representative set of 12 models. Many of the models were built by third parties. Some models were implemented into systems although we did not have access to some of the implemented systems. Some models we reverse engineered from the implemented system. The sizes of the models varied substantially, with up to several hundred classes.

In total, considering the large number of models, experiments, and model elements involved, we found that our technique produces reliable results 96% of the time (only 4% false positives). For about two thirds of the experiments, our approach did not produce any false positives. For the remaining one third, our approach produced less than 10% bad results—with one exception: In one case study, our approach produced 40% incorrect results. This is a very high number but given the small size of the model (26 classes), higher fluctuations are to be expected. Although our approach produces highly reliable results most of the time, all results have to be investigated to reason about their correctness. Section 6.2 (manual versus automated abstraction) discusses that it is significantly cheaper to manually inspect abstraction results produced by our approach instead of abstracting manually.

Our rules are tailored in a fashion that prevents false negatives. This implies that the lack of an abstraction truly means that no abstraction exists. Because of this, our abstraction technique has 100% sensitivity. Note, sensitivity refers to the proportion of paths that are abstractable that have positive abstraction results. It is computed as (True Positives)/(True Positives+False Negatives). Our abstraction technique also has 99.3% specificity. Specificity refers to the proportion of paths that are not abstractable that have no abstraction results (True Negatives)/(True Negatives+False Positives). True negatives were computed by subtracting abstracted paths from all investigated paths. For more details about these metrics, please refer to [Egy02].

6.2 Manual Abstraction versus Automation

Despite our approach's preference to err in favor of abstracting too much instead of too little, it produces mostly correct abstraction results. This has the advantages that the designer does not get overwhelmed with too much (wrong) information and consequently incorrect abstraction results can be identified with reasonable effort. We observed during validation that our approach produced a total of 418 abstract relationships among 170 abstract classes (ratio of 2.45 relationships per class). This is not much higher than the ratio among original relationships and original classes which is 1.83.

The low number of false positives produced by our approach also implies that it is significantly easier to validate abstraction results produced by our approach than having to abstract paths manually. It still requires a human decision maker to make the final judgment on the correctness of the abstraction result but our approach relieves the human designer from the extremely time consuming task of inferring abstract relationships among all class combinations and potential paths. We observed that there were 21024 potential dependencies among all 170 abstract classes of all 18 experiments; but there were only 258 transitive relationships. It follows that there were almost 100 times more dependencies to investigate than transitive relationships to validate. Even if tool support is provided that automatically determines all paths among abstract classes, we found that there were 2374 different paths among the 170 abstract classes. Most of those paths were not abstractable and thus there was still a 10-fold benefit in manually validating the abstraction results versus manually abstracting those paths. This data showed clearly that it is significantly better to investigate the abstracted diagrams without having to do all the abstraction work. The task of validating abstraction results is additionally simplified through trace information (mapping) between abstraction results and their original input. This makes it straightforward for designers to trace back particular abstraction results to investigate their origin and consequently their correctness. As for the actual effort required for validating results, this is entirely dependent on the designer's familiarity with the models. We found that if a designer is very familiar with a model then it was generally straightforward and fast to judge the correctness of abstraction results. For more details about these metrics, please consult [Egy02].

7 RELATED WORK

Many techniques have been proposed to aid the understanding of complex class diagrams. There are reading techniques such as inspection [Fag86] that use group effort to cope with complexity. Most of these techniques are manual and involve high effort and manpower. Using multiple views is an effective form of separating concerns [TOHS99]. Class diagrams can be subdivided into multiple views [AHM88][FKN+91][Gar88] where partial and potentially overlapping portions of the diagram are depicted. The sum of all views (diagrams) is the complete class diagram itself. Multiple views make use of the fact that one does not need access to all classes to understand a particular concern. Although multiple views can make classes belonging to individual concerns more understandable, they generally do not project a high-level, simplified abstraction of the overall class diagram.

Lieberherr et al. [LHX94] defined class transformation methods to capture evolution. They argue that class evolution is inevitable and results in new class models that, preferably, should be as consistent as possible with earlier versions. Although, one could argue that evolution is a form of refinement, we take a more narrow stance. For us, refinement has to maintain consistency within a given model. Their work thus addresses evolutionary 'refinement' and 'consistency issues' that are considered outside the scope of this chapter. Nonetheless, one can envision a strong need for our approach to be combined with theirs so that model refinement and abstraction can be complemented with model evolution.

Fahmy and Holt [FH00] examined structural aspects of models in form of graph re-writing. In their work, they define rules on how to transform graph patterns. They do not single out class diagrams; however, their work is applicable since class diagrams can be seen as graphs containing vertices (classes) and edges (relationships). They also define transformation rules for 'lifting' and 'hiding interior/exterior' which could be seen analogous to our approach. Indeed, graph re-writing could provide a more generic framework for our work and we are considering to integrate some of their ideas; however, currently they do not define class abstraction rules in the level of detail we do, nor do they define an algorithm that can avoid problems of race conditions. Furthermore, their transformation algorithm is computationally very expensive since they can define complex patterns and anti-patterns. Instead, our approach relies on relatively simple patterns that can be abstracted quickly.

The works of Schuerr et al. [SWZ95] is similar to Fahmy and Holt. They also propose a graph rewriting approach called PROGRES with similar limitations. However, an interesting feature of PROGRES is the improved performance of pattern matching which they recognized as being a severe problem. They propose a heuristic-based approach that optimizes the use of a limited set of graph rewrite rules to achieve faster performance. The limitation of their approach is that it works best on small sets of rules. We took an alternative approach with a large number of graph rewrite rules (abstraction rules) but only very simple rule patterns (string of relationships). Our pattern matching approach is thus as simple and as efficient as string matching. As such, we see their work as an interesting alternative in dealing with the computationally expensive problem of pattern matching.

Streckenbach and Snelting [ST98] devised a technique in restructuring class hierarchies by investigating how classes are used by applications. In a form they abstract the essence of classes by creating perspectives of class hierarchies as relevant to individual applications. They then combine those individual perspectives to yield a better class hierarchy. Although their work re-interprets class diagrams (hierarchies) it cannot be used to reason about abstract interdependencies among classes. It is, however, a good example that class hierarchies (or diagrams) are ambiguous and information within them (i.e., methods) can be moved around without destroying behavioral consistency.

Racz and Koskimies [RK99] created an approach to class abstraction that is probably the closest to ours. They also recognized the powerful but simple nature of abstracting relationships with classes into abstract relationships. However, they only defined a small set of abstraction rules and they did not investigate the issue of path abstraction. As a result, they did not devise an automatable abstraction technique but instead developed a tool for semi-automated use. In Section 6, we pointed out the disadvantages of semi-automated abstraction on large-scale class diagrams. Irrespective of the drawbacks of their approach, we see their work as a confirmation of the validity of our abstraction technique because like us they acknowledge the usefulness of abstracting class patterns based on the transitive meaning of relationships.

Our abstraction technique is conceptually related to transformation techniques such as Sequence to Statechart transformation [KSTM98][SKK01], Collaboration to Statechart transformation [KEK98], and Sequence to Class transformation [TE00]. All these approaches recognized the fact that model transformation in general can be done without the use of intermediate, third-party languages. For instance, [KSTM98] describes an approach for combining sequence diagrams into statechart diagrams directly without creating the overhead of using an additional languages. These works demonstrate that it is possible to define precise, formal transformations using informal languages (UML diagrams) as input and generating other informal languages as output. Our approach is also well-defined and formal and like their approaches we avoided using third-party languages to represent UML although such languages exist.

8 CONCLUSION

This chapter presented an approach for the automated abstraction of class diagrams. The approach investigated the semantic meaning of paths of classes to infer transitive properties. Although our abstraction rules are primitive in form, they are rich enough in number to abstract large-scale class diagrams. To date we have validated the technique and its rules on numerous third-party applications and models with up to several hundred model elements. We showed that our technique scales and produces correct results most of the time. We demonstrated various forms of ambiguities and showed that there are ways of living with them even preserving them during transformation.

Our abstraction approach is fully tool supported and integrated with IBM Rational Rose^{TM}. We believe our abstraction technique to be well-suited for model understanding, reverse engineering, and consistency checking. During model understanding, our technique provides a light-weight, fast, and easy to use method for 'zooming out' of a model for inspection (e.g., whenever the model changes). For reverse engineering, our technique helps in creating higher-level interpretations of implementation classes and their relationships. And for consistency checking, our approach makes a lower-level class diagram easier comparable with existing higher-level ones.

9 ACKNOWLEDGEMENTS

We wish to thank Philippe Kruchten for the initial idea and support. We also wish to thank Barry Boehm, Cristina Gacek, Paul Grnbacher, Nenad Medvidovic, Dave Wile, and the anonymous reviewers for insightful discussions. This work was supported by Rational Corporation and DARPA through contracts F30602-99-1-0524, F30602-00-C-0200, and F30602-00-C-0218.

References

[AAHK99] M. Abi-Antoun, J. Ho, and J. Kwan. *Inter-Library Loan Management System: Revised Life-Cycle Architecture*. Center for Software Engineering, University of Southern California, Los Angeles, CA, USA, 1999.

[AHM88] R.A. Altmann, A.N. Hawke, and C.D. Marlin. An integrated programming environment based on multiple concurrent views. *Australian Computer Journal*, 20:65–72, 1988.

[Alv98] S. Alvarado. An evaluation of object oriented architecture models for satellite ground systems. In *Proceedings of the 2nd Ground Systems Architecture Workshop (GSAW), El Segundo, CA*. 1998.

[BRJ99] G. Booch, J. Rumbaugh, and I. Jacobson. *¡The Unified Modeling Language User Guide*. Addison Wesley, 1999.

[EFLR98] A. Evans, R. France, K. Lano, and B. Rumpe. The uml as a formal modeling language. *Journal Computer Standards & Interfaces*, 19, 1998.

[Egy02] A. Egyed. Automated abstraction of class diagrams. *ACM Transaction on Software Engineering and Methodology (TOSEM)*, 11:449–491, 2002.

[Egy03] A. Egyed. A scenario-driven approach to trace dependency analysis. *IEEE Transactions on Software Engineering (TSE)*, 29:116–132, 2003.

[EK99] A. Egyed and P. Kruchten. Rose/architect: A tool to visualize architecture. In *Proceedings of the 32nd Hawaii International Conference on System Sciences (HICSS), Maui, HI*. 1999.

[EW01a] A. Egyed and D. Wile. Statechart simulator for modeling architectural dynamics. In *Proceedings of the 2nd Working International Conference on Software Architecture (WICSA), Amsterdam, The Netherlands*, pages 87–96. 2001.

[EW01b] A. Egyed and D. Wile. Statechart simulator for modeling architectural dynamics. In *Proceedings of the 2nd Working International Conference on Software Architecture (WICSA), Amsterdam, The Netherlands*, pages 87–96. 2001.

[Fag86] M.E. Fagan. Advances in software inspections. *IEEE Transactions on Software Engineering (TSE)*, 12:744–751, 1986.

[FH00] H. Fahmy and R.C. Holt. Using graph rewriting to specify software architectural transformations. In *Proceedings of the 15th IEEE International Conference on Automated Software Engineering (ASE)*, pages 187–196. 2000.

[FKN+91] A. Finkelstein, J. Kramer, B. Nuseibeh, L. Finkelstein, and M. Goedicke. Viewpoints: A framework for integrating multiple perspectives in system development. *International Journal on Software Engineering and Knowledge Engineering*, 31-58, 1991.

[Gar88] D. Garlan. Views for tools in integrated environments. In *Advanced Programming Environments*, pages 314–343. 1988.

[GF94] O.C.Z. Gotel and A.C.W. Finkelstein. An analysis of the requirements traceability problem. In *Proceedings of the First International Conference on Requirements Engineering*, pages 94–101. 1994.

[GGZ+05] Lars Grunske, Leif Geiger, Albert Zündorf, Niels Van Eetvelde, Pieter Van Gorp, and Daniel Varro. Using graph transformation for practical model driven software engineering. In Sami Beydeda, Matthias Book, and Volker Gruhn, editors, *Model-driven Software Development –*

Volume II of Research and Practice in Software Engineering. Springer, 2005.

[GZL05] Jeff G. Gray, Jing Zhang, and Yuehua Lin. Generic and domain-specific model refactoring using a model transformation engine. In Sami Beydeda, Matthias Book, and Volker Gruhn, editors, *Model-driven Software Development – Volume II of Research and Practice in Software Engineering*. Springer, 2005.

[Ham05] Imed Hammouda. A tool infrastructure for model-driven development using aspectual patterns. In Sami Beydeda, Matthias Book, and Volker Gruhn, editors, *Model-driven Software Development – Volume II of Research and Practice in Software Engineering*. Springer, 2005.

[JR00] D. Jackson and M. Rinard. Software analysis: A roadmap. In *Proceedings of the 20th International Conference on Software Engineering (ICSE)*, pages 133–145. 2000.

[KEK98] I. Khriss, M. Elkoutbi, and R. Keller. Automating the synthesis of uml statechart diagrams from multiple collaboration diagrams. In *Proceedings for the Conference of the Unified Modeling Language*, pages 132–147. 1998.

[KSTM98] K. Koskimies, T. Syst, J. Tuomi, and T. Mnnist. Automated support for modelling oo software. *IEEE Software*, 87-94, 1998.

[LHX94] K.J. Lieberherr, W.L. Hursch, and C. Xiao. Object-extending class transformations. *Journal Formal Aspects of Computing*, 6:391–416, 1994.

[MC01] W.E. McUmber and B.H.C. Cheng. A general framework for formalzing uml with formal languages. In *Proceedings of the 23rd International Conference on Software Engineering (ICSE)*, pages 433–442. 2001.

[Met05] Andreas Metzger. A systematic look at model transformations. In Sami Beydeda, Matthias Book, and Volker Gruhn, editors, *Model-driven Software Development – Volume II of Research and Practice in Software Engineering*. Springer, 2005.

[Ö98] G. Övergaard. A formal approach to relationships in the unified modeling language. In *Proceedings of the Workshop on Precise Semantics for Software Modeling Techniques (PSMT98)*, pages 91–108. 1998.

[RK99] F.D. Racz and K. Koskimies. Tool-supported compression of uml class diagrams. In *Proceedings of the 2nd International Conference on the Unified Modeling Language (UML)*, pages 172–187. 1999.

[SKK01] S. Schönberger, R.K. Keller, and I. Khriss. Algorithmic support for model transformation in object-oriented software development. *Concurrency and Computation: Practice and Experience*, 13:351–383, 2001.

[ST98] G. Snelting and F. Tip. Reengineering class hierachies using concept analysis. In *Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 99–110. 1998.

[SWZ95] A. Schuerr, A.J. Winter, and A. Zündorf. Graph grammar engineering with progres. In *Proceedings of the 5th European Software Engineering Conference (ESEC)*, pages 219–234. 1995.

[TE00] A. Tsiolakis and H. Ehrig. Consistency analysis of uml class and sequence diagrams using attributed graph grammars. In *Proceedings of GRATRA 2000, Berlin, Germany*, pages 77–86. 2000.

[TOHS99] P. Tarr, H. Osher, W. Harrison, and S.M.Jr. Sutton. N degrees of separation: Multi-dimensional separation of concerns. In *Proceedings of the*

21st International Conference on Software Engineering (ICSE 21), Los Angeles, CA, pages 107–119. 1999.