Automated Abstraction of Class Diagrams

ALEXANDER EGYED
Teknowledge Corporation

Designers can easily become overwhelmed with details when dealing with large class diagrams. This article presents an approach for automated abstraction 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, can abstract complex class structures quickly. This article presents those abstraction rules and an algorithm for applying them. 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.

Categories and Subject Descriptors: D.2.2 [**Software Engineering**]: Design Tools and Techniques

General Terms: Design

Additional Key Words and Phrases: Class abstraction, class diagrams, class patterns, reverse engineering, transformation, unified modeling language

1. INTRODUCTION

In the course of software development, a software system may be refined concurrently in various dimensions. There is the physical refinement of a software system into subsystems, components, packages, classes, methods, and actual lines of code. There is the logical refinement of a software system where a greater level of detail about a given software element is unraveled the closer one looks. And there is the goal-driven refinement of a software system where requirements (small or big) are refined into design or implementation.

Refinement is often considered the "natural course" of software development where a problem is evolved into a solution, but the more a software problem 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 process that transforms lower-level elements into higher-level elements containing fewer details on a larger granularity. Abstraction provides software designers and programmers with the ability to zoom out of diagrams to investigate

This research was supported by Rational Corporation and DARPA through contracts F30602-99-1-0524, F30602-00-C-0200, and F30602-00-C-0218.
Author's address: Teknowledge Corporation, 4640 Admiralty Way, Suite 231, Marina Del Ray, CA 90292; email: aegyed@acm.org.
Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or direct commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 1515 Broadway, New York, NY 10036 USA, fax: +1 (212) 869-0481, or permissions@acm.org.
© 2002 ACM 1049-331X/02/1000-0449 $5.00

them where fewer details mask their interrelationships. Class abstraction has a number of vital uses:

(1) It aids program and model understanding by reducing the number of lower-level elements to the most important, higher-level elements.
(2) It supports consistency validation by comparing existing higher-level models [Egyed 2000] or architectures [Egyed and Medvidovic 2000] with abstracted ones.
(3) It assists reverse engineering by transforming lower-level models into higher-level ones [Mueller et al. 2000].

In essence, the process of abstraction is the simplification of models by removing details not necessary on a higher, more abstract level. This article investigates the problem of abstracting class diagrams (i.e., as defined in the Unified Modeling Language [Booch et al. 1999]) where the traditional perception of abstraction is often seen as the grouping of lower-level elements into higher-level elements (classes are grouped into packages or into higher-level classes) [Fahmy and Holt 2000; Siegfried 1996]. This article will first demonstrate the limitations of grouping as a mechanism for abstraction and it will then present a rule-based technique for automated abstraction of class diagrams that avoids these limitations.

Our abstraction technique uses abstraction rules that have input and result patterns. Abstraction rules define the semantics of how a set of model elements can be replaced by a less complex, more-abstract model element. Our abstraction algorithm then performs syntactic matching of the abstraction rules on the model. Whenever an input pattern of a rule is encountered in the model, then that pattern is replaced by the result pattern of that rule. We require that every abstraction rule has a result pattern that is simpler than its input pattern. It follows that every application of a rule simplifies a given model.

We evaluated our abstraction technique on over a dozen real-world case studies ranging from in-house-developed models to third-party models. Most notably, we used our technique in connection with the Inter-Library Loan System [Abi-Antoun et al. 1999], a part of a Satellite Ground System [Alvarado 1998], C2SADEL to UML integration [Egyed and Medvidovic 2000], Video-On-Demand System [Dohyung 1999], SDS Statechart Simulator [Egyed and Wile 2001], and other projects. The sizes of the models ranged from several dozen to several hundred model elements (see also Section 8). The validation showed that our technique scales and that it produces correct abstractions 96% of the time. Our approach is fully automated and tool supported.

2. HMS EXAMPLE

For illustrative purposes, this article uses a simplified Hotel Management System (HMS) that provides support for reservations, check-in/check-out procedures, and associated financial transactions. Figure 1 shows a potential hierarchical decomposition of the HMS into the two packages *ServicePackage* and *FinancialPackage*. Both packages are further subdivided into classes and relationships describing calling dependencies, inheritance, and part-of

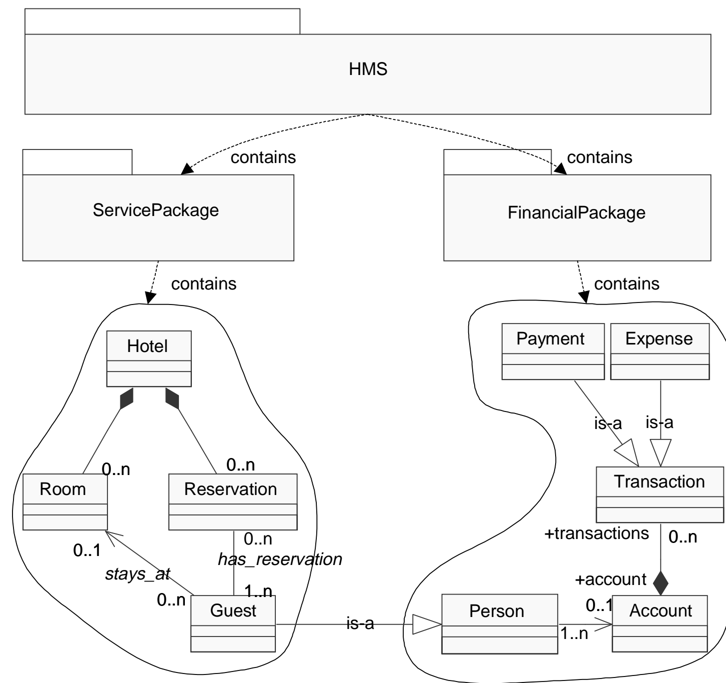

Fig. 1. Tree-like refinement of the HMS from system to packages to classes.

relationships. Figure 1 uses the UML notation [Booch et al. 1999] to describe the three depicted layers of the HMS system (top layer, package layer, and class layer). In particular, UML packages are used to describe the system and packages. UML classes and various types of UML relationships (generalization, association, and aggregation) make up the class layer.

The *FinancialPackage* captures accounts and monetary transactions of people (*Person*). It 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*. The realization of *Person* in form of *Guest* provides a bridge between the generic *FinancialPackage* and the specific, domain-dependent *ServicePackage*. The *ServicePackage* uses the *FinancialPackage* and defines additional 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*.

A note on terminology issues: Given the variety of different but similar terms in the transformation, reverse engineering, and consistency-checking community, we decided to align our terminology according to the UML definition. A "model" is a description of a system (i.e., HMS) and "diagrams" are visualizations of a model. Whereas a model describes the entirety of a system, diagrams

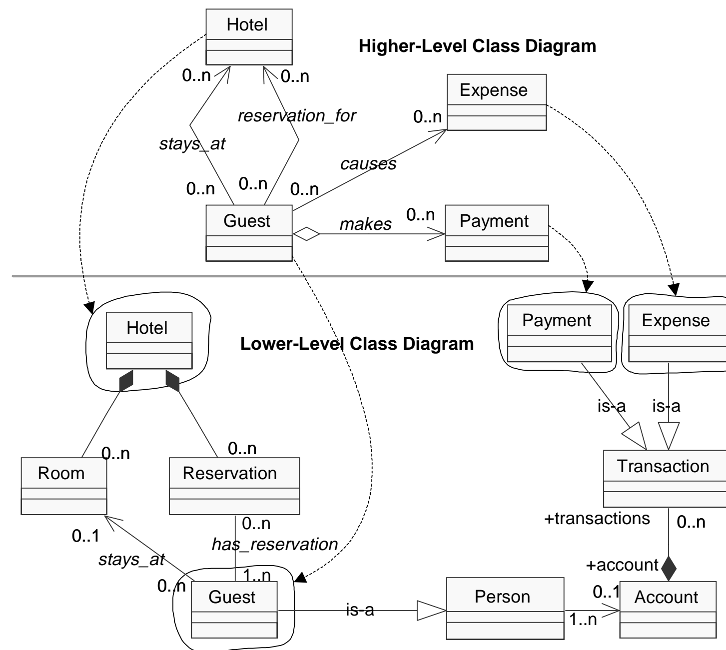

Fig. 2. Logical refinement (traces capture dependencies).

may depict pieces thereof (views). Figure 1 is a diagram of the HMS model and it shows a part of its physical decomposition. For our approach, it is not of significance whether to abstract models or their partial diagrams. Furthermore, the term, *abstraction*, is not to be confused with terms such as *abstract classes* in programming languages that denote class definitions with only partial code.

3. REFINEMENT, ABSTRACTION, AND HIERARCHIES

To understand the need for abstraction, this section discusses three different refinement scenarios that may be encountered during software development—logical refinement, goal-oriented refinement, and physical refinement. This discussion is not meant to be exhaustive and it merely presents refinement scenarios that are frequently observed in real-world situations. These refinement scenarios will be used throughout this article to illustrate (1) why abstraction is needed and (2) why the grouping of classes is insufficient to achieve abstraction.

3.1 Grouping in Context of Logical Refinement

A very common form of refinement is what we call "logical refinement." During logical refinement, details are added to a given set of software elements over time. Logical refinement can be seen analogous to a magnification glass where more information becomes visible the closer one looks. Figure 2 shows class diagrams of the HMS system at two levels of abstraction where the bottom diagram is a refinement of the top diagram. The lower-level class diagram (bottom of Figure 2) is the same as in Figure 1 and reflects our most detailed

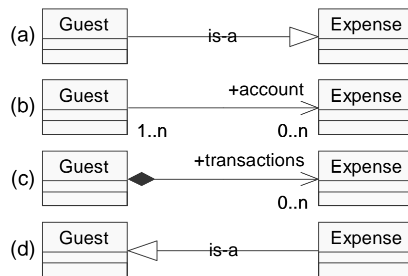

Fig. 3. Four possible abstraction results for lower-level classes.

understanding of how the HMS works. On the other hand, the higher-level diagram in Figure 2 (top) summarizes the lower-level one by omitting information considered less relevant. For instance, using an account to maintain payment and expense transactions could be considered an implementation detail. Naturally, it is in the eye of the beholder which of the two diagrams (higher or lower level ones) is more useful or important. The lower-level diagram makes realization decisions that are relevant for later implementation, but the higher-level diagram is more readable and almost as informative although it uses fewer model elements.

Logical refinement tends to result in one-to-many mappings between single, higher-level elements and one or more lower-level elements. However, lower-level elements cannot always be clearly associated with single higher-level elements. As an example, consider the *Reservation* class between *Guest* and *Hotel*. It is not intuitive to claim that *Reservation* belongs more to *Guest* than to *Hotel* or vice versa. *Reservation* is rather a “contract” that binds *Person* and *Hotel*. For abstraction, this implies a difficulty in determining what to do with “helper classes” such as *Reservation* that may seem less important. Since the notion of important class versus helper class depends on the particular view of the designer, the challenge of abstraction becomes how to determine abstract relationships among important classes without “helper classes” obstructing the view.

Note that, in the lower-level diagram of Figure 2, one cannot infer any direct relationships among *Guest*, *Hotel*, *Expense*, and *Payment* because none exist. Thus, if one were to use grouping as the only means of abstraction, then it would become problematic on how to best group the classes to derive correct, abstract relationships among them. In Figure 3, potential abstract relationships between *Guest* and *Expense* are suggested which were derived by grouping the helper classes (*Person*, *Account*, and *Transaction*) between them in different manners. For instance, if the helper classes *Person*, *Account*, and *Transaction* were to be grouped with *Expense*, then one could derive an abstraction as in Figure 3(a) where *Guest* inherits from *Expense*. Alternatively, if *Person* were to be grouped with *Guest* and *Account* and *Transaction* were to be grouped with *Expense*, then one could derive the abstraction in Figure 3(b). Figures 3(c) and 3(d) show the results for grouping *Person*, *Account*, and *Transaction* in yet other ways. Even if one of the four abstractions were actually correct, it is not clear how an automated process could decide which one that would be. It follows

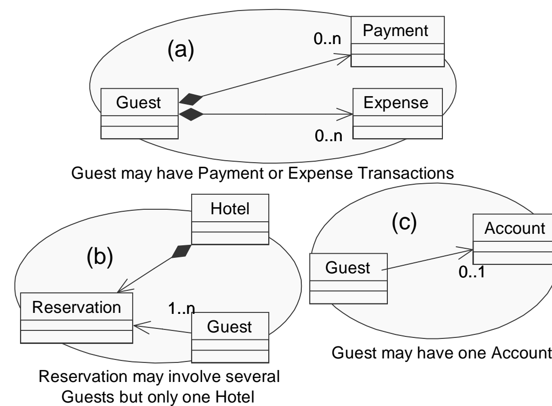

Fig. 4. Goal-driven refinement (see Figure 1 for refinement of above three class structures).

that grouping is not suitable to enable abstraction in this case. We will revisit this problem later.

3.2 Grouping in the Context of Goal-Driven Decomposition

During goal-driven refinement, design information is described as it relates to goals (i.e., requirements). Goal-driven refinement singles out subsets of classes for specific tasks. For instance, Figure 4(a) describes a use case (task) that states that a *Guest* may have *Expense* transactions (e.g., staying at *Hotel*) and *Payment* transactions (e.g., paying for room). For that task, one needs to be aware of classes such as *Guest*, *Expense*, and *Payment* which are a "slice" across the physical structure in Figure 1 [Snelting and Tip 1998]. Goal-driven refinement thus creates logical structures associated with physical structures based on some needs. Naturally, this implies that there are potentially many logical ways of decomposing a system. Figure 4 shows three "slices" across the physical structure in Figure 1 (note that Figure 1 is the refinement for all three slices and omitted in Figure 4).

For abstraction, this implies that there are many ways of abstracting a class diagram. The slice in Figure 4(a) is a true subset of the higher-level diagram presented in Figure 2, but an abstraction of the lower-level diagram. The slice in Figure 4(b) is a true subset of the lower-level diagram in Figure 2, and the slice in Figure 4(c) is a new interpretation neither visible in Figure 1 nor in Figure 2. Figure 4(c) states that a *Guest* may have zero or one *Accounts*. In the higher-level diagram, the class account was considered too detailed and was omitted and, in the lower-level diagram, the class *Person* was added, which obscured the relationship between *Guest* and *Account*. Conceptually, we could think of Figure 4(c) as a partial refinement of the higher-level diagram and a partial abstraction of the lower-level diagram. A developer thus might be interested to know whether the addition of *Person* changed anything about the required relationship between *Guest* and *Account* (consistency). Again, the grouping of classes is not very useful for abstraction. If the classes in Figure 1 were to be abstracted to the perceived relationships in Figure 4, then grouping would lead

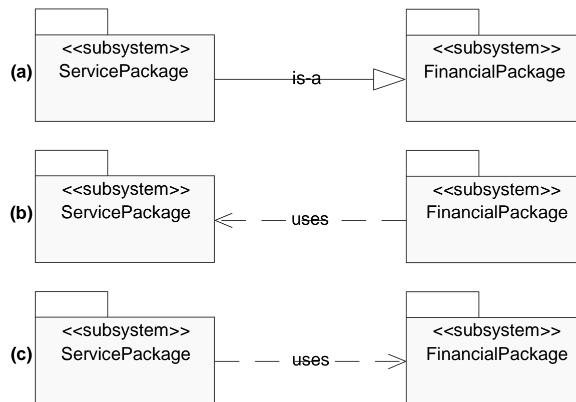

Fig. 5. Three possible abstraction results for lower-level classes.

to the same problems as were described in Section 3.1. Later, it will be shown how our abstraction technique handles this situation.

3.3 Grouping in the Context of Physical Refinement

Figure 1, discussed earlier, depicted the physical refinement of the HMS into packages and classes. Figure 1 is an example of a tree-structured hierarchy that results from refining systems into packages (subsystems), and packages into classes using a one-to-many mapping over several levels of refinements. This form of refinement is simply achieved by subdividing higher-level elements into one or more lower-level elements. Given the simple one-to-many mapping during physical refinement, abstraction may be seen as the grouping of nonoverlapping sets of lower-level elements into individual, higher-level elements until the desired granularity is reached [Racz and Koskimies 1999]. Nonetheless, even in context of physical refinement the grouping of classes may be inadequate to achieve abstraction since relationships among grouped classes may not map easily to relationships among packages. Consider the inheritance relationship between classes *Guest* and *Person*, which is the only relationship that spans across the two packages (Figure 1). Clearly, it is wrong to "raise" interdependencies established on the class level to the package level. Figure 5(a) shows the result of grouping the classes belonging to packages and using the relationships among classes as relationships among their packages (i.e., generalization between *Guest* to *Person*). The given result is wrong since *ServicePackage* does not inherit the *FinancialPackage* just because *Guest* inherits from *Person*.

Alternatively, Fahmy and Holt [2000] assume that lower-level elements have to be reinterpreted in the language of the higher-level elements during abstraction. For instance, they define a "method call" relationship among classes to be abstractable to a "uses" relationship among packages with the same directionality. Consequently, they would infer that either *FinancialPackage* uses *ServicePackage* (Figure 5(b)) and/or *ServicePackage* uses *FinancialPackage* (Figure 5(c)). However, in the given example, we find that a generalization relationship and not a method call relationship (i.e., association) is

the bridge between *ServicePackage* and *FinancialService*. How could one interpret the generalization between *Guest* and *Person* in context of the "uses" relationship? It is invalid to conclude that *ServicePackage* uses *FinancialPackage* simply because *Guest* inherits from *Person*. For example, if some class "Foo" in *FinancialPackage* calls *Person*, then polymorphism during object-oriented programming would make it possible that class Foo could call *Guest* (*FinancialPackage* would use *ServicePackage* as in Figure 5(c)). We will discuss in the next section how our abstraction technique solves this problem.

4. SEMANTIC ABSTRACTION

The main goal of abstraction is to hide information from a lower-level diagram that is perceived as not relevant from a higher-level perspective. Since a model can be abstracted in any arbitrary manner, it follows that an abstraction technique needs to be guidable. Guidance may be as simple as a user (i.e., software programmer, designer, or architect) selecting model elements that are of particular interest or guidance may also be given automatically via trace dependency information [Egyed 2001; Gotel and Finkelstein 1994]. In the following, we presume that such guidance is available. The goal of abstraction is then to hide all lower-level elements that were not selected from or traced to so that the resulting abstraction depicts only the desired elements and their perceived types, attributes, methods, and interdependencies that summarize the now-hidden information.

The main challenge of abstraction is that hiding information alone is not sufficient to transform lower-level elements into higher-level elements. Instead, the hidden information has to be reinterpreted in the context of the remaining, nonhidden elements. For instance, if we were to hide the lower-level class *Reservation* and its relationships in Figure 2, then the abstraction would lose the knowledge that *Guest* may have a *reservation_for Hotel*. An abstraction process therefore needs to hide *Reservation* but add a relationship that reinterprets the now-hidden information (class *Reservation*) and its relationships. Given the limitations of grouping (as discussed in Section 2), our technique will instead demonstrate that class patterns can be abstracted in a manner that uses semantic information in diagrams. Our abstraction technique still uses a form of grouping as a method to hiding information but instead of grouping classes our method groups collections of classes with their relationships into higher-level relationships. Simply speaking, our technique uses rules that define how lower-level patterns of classes and relationships can be reinterpreted as higher-level relationships.

This sections will present generic abstraction rules (patterns) that are based on the UML notation for class diagrams [Booch et al. 1999]. Currently supported are class diagrams with the 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. Users of our approach are not required to extend or modify this rule set unless they wish to fine tune it (e.g., domain specific rules).

4.1 Semantic Rules

Revisiting Figure 4 (scenario 3), we are given a requirement that describes the interdependency between the classes *Guest* and *Account*. This requirement is realized in the lower-level diagram (Figure 2) where it is described that a *Guest* is a type of *Person* and *Person* may have zero or one *Accounts*. If a developer would like to make sure that the lower-level diagram does indeed realize the requirement correctly (consistency), then he/she would need to replace the class *Person* and its two relationships to *Guest* and *Account* with a higher-level relationship that transitively summarizes the replaced elements. A developer is thus seeking an abstract relationship that hides *Person*. To find out whether there is indeed such an abstract relationship, we need to analyze the semantic meaning of the relationships among *Guest*, *Person*, and *Account*.

The information that a *Person* may have an *Account* (association relationship) 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 *Accounts*. This knowledge of the transitive relationship between *Guest* and *Account* 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 helper 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 may be used to collapse 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 below).

```
{2} GeneralizationRight - Class - DependencyRight ⇒ DependencyRight
{3} GeneralizationRight - Class - AssociationRight ⇒
    AssociationRight
```

```
{4} GeneralizationRight - Class - [Agg]AssociationRight ⇒
    [Agg]AssociationRight
{5} GeneralizationRight - Class - DependencyLeft ⇒ DependencyLeft
{6} GeneralizationRight - Class - AssociationLeft ⇒ AssociationLeft
{7} GeneralizationRight - Class - [Agg]AssociationLeft ⇒
    [Agg]AssociationLeft
{8} GeneralizationRight - Class - Association ⇒ Association
```

UML class relationships are usually unidirectional, requiring us to differentiate "Left" from "Right." The only exception is the association relationship, which may also be bidirectional. Rule {8} in the above block of patterns therefore states that the bidirectionality 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 the lower-level diagram in Figure 2. 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 2. In the higher-level diagram, it is shown that a *Guest* may have *reservation_for* a *Hotel*. The lower-level diagram then uses the helper class *Reservation* to further refine the *reservation_for* relationship. For abstraction, this implies that one needs to replace the class *Reservation* and its relationships to *Guest* and *Hotel* with a more abstract interpretation. In order to do that, it is again necessary to investigate the transitive meaning of the to-be-replaced model elements. The lower-level diagram shows the class *Hotel* with an aggregation relationship from *Reservation* to *Hotel* implying that *Reservation* is a part of *Hotel*. The lower-level 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 can be applied to other relationships (e.g., rules {13}–{17}):

```
{12} Association - Class - Association [Agg] ⇒ Association
{13} AssociationRight - Class - Association [Agg] ⇒ AssociationRight
{14} AssociationLeft - Class - AssociationLeft[Agg] ⇒ AssociationLeft
{15} AssociationLeft[Agg] - Class - AssociationLeft[Agg] ⇒
     AssociationLeft[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 relationship between *Person* and *Transaction* in lower-level diagram in Figure 2).

4.2 Living with Ambiguous Model Definitions

The example of determining the relationship between *Person* and *Transaction* (lower-level diagram in Figure 2) 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* only relates to a subset of *Account*—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 on 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*. But is this statement always correct? Imagine that *Guest* inherits method “foo” but overwrites its body so that it does not create an instance of class *Account* nor calls the overwritten method of the parent. 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 like these naturally cause a dilemma. On the one hand, we are opposed to using abstraction rules that are not 100% reliable; however, on the other hand, 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 user make decisions in case of uncertainty (e.g., Racz and Koskimies [1999]). Given the large and complex nature of models, semi-automated abstraction can become 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 [Egyed 2000] (see also Section 8). We thus find it unsatisfactory to restrict our abstraction process to semi-automated use. 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 user being aware of it.

UML class diagrams, like many other graphical description languages, were not defined completely precise and unambiguous [Jackson and Rinard 2000]. Indeed, we find that their relaxed nature often encourages their use since software developers are sometimes either unable or unwilling to make precise design decisions. For instance, in UML, it cannot be modeled whether or not 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) [Evans et al. 1998; Övergaard 1998; McUmber and Cheng 2001] but their use is generally optional and left to the discretion of the designer.

Since the basic notation of UML is ambiguous, we took the stance that automated abstraction needs to be able 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 "null" (no relationship) in other cases. This implies that abstract relationships indicated by our approach may or may not factually exists. In cases where more complex abstractions allow multiple abstract interpretations, our approach will suggest all of them and indicate this uncertainty (ambiguity) in form of an annotation ("and/or" label). Our solution has the advantage that no abstract results are omitted although false positives may happen. Section 8 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.

4.3 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.

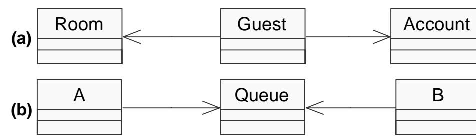

Fig. 6. Abstracting association relationships.

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 unidirectional association is abstracted together with a bidirectional association, the bidirectionality 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`). Association relationships are rather straightforward to abstract, except if they are counter directional. Figure 6 depicts the two scenarios of counter-directional association relationships.

If a helper class has two outgoing association relationships (e.g., *Guest* in Figure 6 (a)), then no interaction is possible (e.g., *Room* and *Account* cannot interact). Abstracting "`AssociationLeft - Class - AssociationRight`" results in no relationship. The situation is much less simple in case two associations terminate in a helper class (e.g., *Queue* in Figure 6(b)). One could argue that it is not possible for class A to access methods of class B and for class B to access methods of class A since no transitive calling dependency exists but what about data dependency? For instance, it is possible that class A stores data in the queue which is then read by class B. Interestingly, UML class diagrams do not have relationship types for data dependence. Therefore, no abstraction exists for the pattern "`AssociationRight - Class - AssociationLeft`."

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, that is, 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 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.

On a final note, abstraction rules for generalization relationships are very similar to those of aggregation relationships. It is our finding that the transitive meaning of aggregation and generalization in context of class diagrams is close because one could perceive the parent class as a part of the child class assuming that no methods get overwritten. A noteworthy exception is the "`[Agg]AssociationRight - Class - AssociationLeft`" pattern that is not

abstractable because it states that only the whole can access the part and not vice-versa. On the other hand, the similar pattern based on generalization "`GeneralizationRight - Class - AssociationLeft`" is abstractable because of object-oriented polymorphism.

4.4 Complete List of Abstraction Rules

To date, we have validated our approach in context of UML class structures and the relationship types generalization, association, dependency, and aggregation. Considering directionality, this implies eight unidirectional relationship types such as `GeneralizationRight` or `AggregationLeft` plus three bidirectional relationship types `Association`, `[Agg]Assocition`, and `Association[Agg]`. Altogether, those relationships may form 121 different patterns (11*11). Some of those patterns are abstractable (e.g., "`Association - Class - AggregationRight`") while other patterns are not abstractable (e.g., "`GeneralizationRight - Class - GeneralizationLeft`"). Table I 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 - Generalization Right`" (rule 1) is equvalent to the pattern "`GeneralizationLeft - Class - GeneralizationLeft`" (rule 16).

5. COMPOSITE ABSTRACTION

The previous section discussed abstraction in context of numerous simple rules. Every abstraction rule in itself is not very powerful, but this section will demonstrate that complex class structures can be abstracted by using those rules together. The following will discuss the serial and parallel application of abstraction rules. Serial abstraction describes how to remove a sequence of helper classes. Parallel abstraction describes how to reconcile results of alternative abstraction options (paths). This section will also describe several special cases.

5.1 Serial Abstraction

Abstraction rules can be serialized to abstract a sequence of classes. Consider Figure 7 where, on the upper left, an association relationship between *Person* and *Account* is depicted. It is furthermore stated that *Guest* is a child of *Person* and that *Transactions* are part of *Account*. If it is of interest to know the more abstract relationship between *Guest* and *Transaction* then the abstraction rules in Table I may be applied in sequence to eliminate both helper classes *Person* and *Account*.

For instance, Rule 3 (Table I) may be used to remove *Person* and its relationships to *Guest* and *Account*; and to add a more abstract association relationship from *Guest* to *Account* directly (Figure 7 upper-right). Alternatively, rule 54 may be used to eliminate *Account* (Figure 7 lower-left). Both abstraction

Table I. 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

continued

Table I. *Continued*

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

continued

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

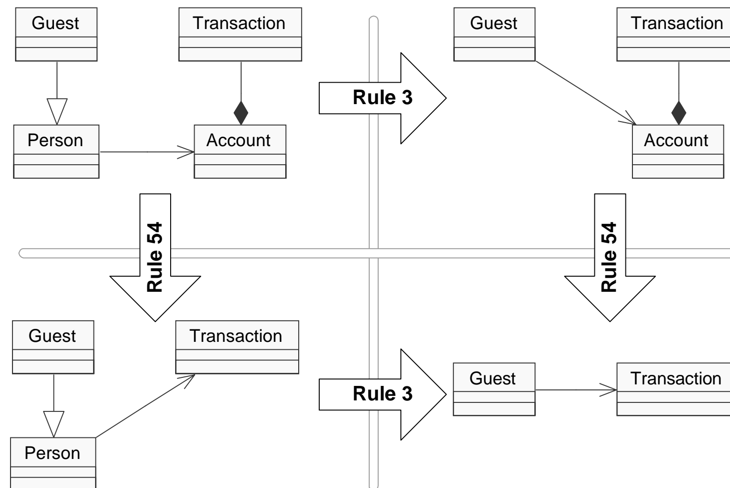

Fig. 7. Symmetric serial abstraction; Same abstraction results produced although rules are applied in different order.

scenarios result in less complex, more abstract class diagrams that contain less classes. However, further abstraction is necessary since the desired result (elimination of both *Guest* and *Account*) has not been reached. Since no rule has precedence over any other rule, both partially abstracted class diagrams need to be abstracted further. Interestingly, we observed that most of our rules are closed in that now the reverse set of rules is necessary to continue abstraction. For instance, in the case where rule 3 was applied first, rule 54 needs to be

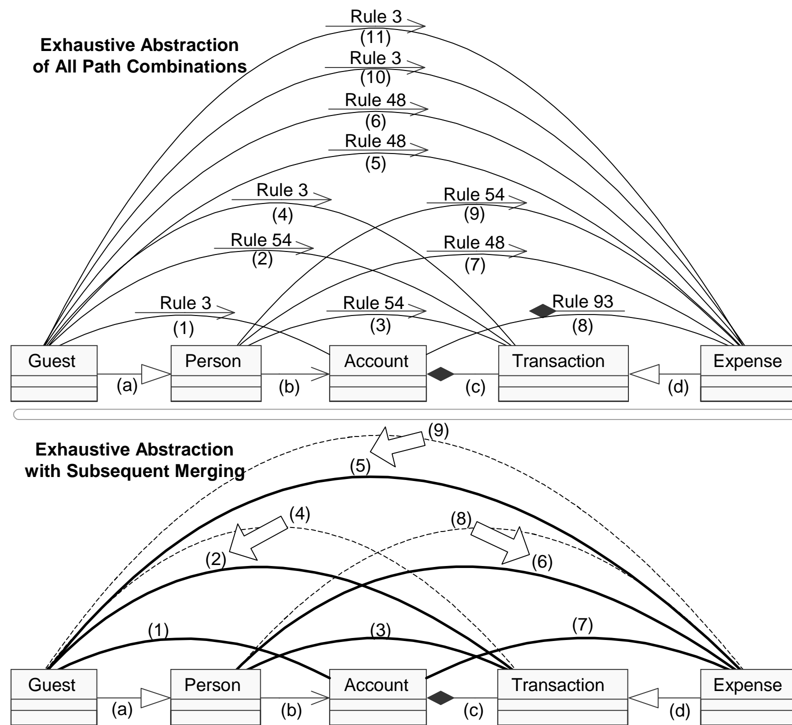

Fig. 8. Serial abstraction of long paths; investigation of all abstraction combinations yields results that can be merged significantly reducing the computational complexity of abstraction.

applied next. Similarly, in the case where rule 54 was used initially, rule 3 is applied next. In both cases, the final abstraction is an association from *Guest* to *Transaction*

We refer to a sequence of helper classes between two important classes as a *path*. Two different combinations for the same path were explored in Figure 7 and two equal abstraction results were generated. Those results can be merged since a single path from one class to another class should only have a single abstract interpretation. The resulting abstraction in Figure 7 is therefore a single association between *Guest* and *Account*. If the two abstraction results would have differed (e.g., different relationship type), then they could not be merged and both results would have to be maintained (ambiguity). This implies that all combinations of paths have to be evaluated in order to determine abstractability. This is necessary because our simple abstraction rules have a narrow perspective on classes and their relationships. By investigating all combinations of abstraction rules, one essentially investigates all possible perspectives in which classes could be used together.

5.2 Serial Abstraction of Long Paths

Serial abstraction of long paths is possible through the repeated use of the simple abstraction rules in Table I. Figure 8 shows the path between the

"important" classes *Guest* and *Expense* via the helper classes *Person*, *Account*, and *Transaction*. In order to abstract such a long path for which no abstraction rule exists, the path is broken down into smaller pieces followed by abstracting those pieces and uniting their results. As was motivated previously, it is necessary to explore all abstraction combinations in order to not favor some abstraction rules over others. Figure 8 (top) shows that initially the subpaths *Guest-Person-Account*, *Person-Account-Transaction*, and *Account-Transaction-Expense* are abstracted. The results of those abstractions are then abstracted together with other elements of the path. For instance, the result of abstracting *Account-Transaction-Expense* is an aggregation from *Expense* to *Account* (rule 93), which is then abstracted as *Person-Account-Expense* to yield an association from *Person* to *Expense* (rule 54).

In order to reduce the potentially large number of abstraction results to a minimum, alike results are merged. Figure 8 (bottom) shows that the abstraction of the subpaths *Guest-Person-Account* and *Person-Account-Transaction* (labeled (1) and (3) in Figure 8) are abstracted separately into distinct interpretations for the subpath *Guest-Person-Account-Transaction*. In particular, abstraction (1) together with the aggregation to *Transaction* (labeled (c) in Figure 8) yields abstraction (2). Similarly, abstraction (3) together with the generalization to *Guest* (labeled (a)) yields abstraction (4). Since both abstraction (2) and (4) cover the same subpath (*Guest-Person-Account-Transaction*) and since both abstractions are of the same type ("Association") and directionality ("`AssociationRight`"), it is legal to merge them. The rationale is as follows: Regardless of how one looks at the (sub) path, there ought to be an association relationship. The final result of abstracting the path *Guest*-to-*Expense* is a single, unambiguous, unidirectional association relationship from *Guest* to *Expense*.

Note that the computational overhead of our abstraction approach may seem very large. This is not the case because of automated reuse and merging of abstraction results discussed later in this section. Sections 8 will also show empirical measurements about the performance of our approach.

5.3 Serial Abstraction with Parallel Paths

Serial abstraction works well in context of a single string of classes. However, in class diagrams, there are often multiple paths of classes between two important classes. For instance, there are two paths that lead from the class *Guest* to the class *Hotel* (lower-level diagram in Figure 2; see also top of Figure 9). One paths includes the bidirectional association between *Guest* and *Reservation* and the aggregation from *Reservation* to *Hotel*; the other path includes the unidirectional association from *Guest* to *Room* and the subsequent aggregation from *Room* to *Hotel*. Our approach evaluates each path separately to determine its abstraction. In Figure 9, rule 97 can reduce the *Hotel-Reservation-Guest* pattern to a simple bidirectional association between *Guest* and *Hotel*. Furthermore, rule 95 can reduce the *Hotel-Room-Guest* pattern to a simple unidirectional association from *Guest* to *Hotel*.

Given that both paths use mutually exclusive model elements, one might assume that they describe separate, abstract relationships. This is certainly

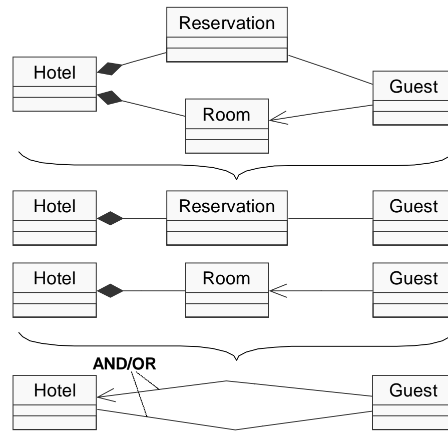

Fig. 9. Serial abstraction with parallel paths leads to separate but related abstraction results.

true in the example in Figure 9, but sometimes a lower-level design may split a higher-level relationship into separate lower-level elements. It is not possible to automatically determine whether to merge abstraction results of parallel paths, whether to declare them ambiguous, or whether to leave them separate. We thus state that *Guest* and *Hotel* are either connected by a bidirectional association, or a unidirectional association from *Guest* to *Hotel*, or both of the above (AND/OR ambiguity). This uncertainty is captured as additional metainformation as can be seen in Figure 9 (bottom).

We have not encountered a case where our rule set produces different abstract relationships for a single abstraction path. However, since we allow users of our approach to extend or manipulate the given set of rules (Table I) it is possible that even the resolution of a single path may lead to ambiguities. These ambiguities would be handled analogously to the discussion in this section (maintain all results and declare them ambiguous via the "AND/OR" annotation).

5.4 Resolving Some Ambiguities during Serial Abstraction

Although individual abstraction steps may cause ambiguities, those ambiguities may not necessarily lead to ambiguous, final abstraction results. This is especially true when long paths are abstracted. Figure 10 shows one such example in context of abstracting the paths between *Hotel* and *Account*. Since this example is an extension of Figure 9, we encounter an ambiguity because of the parallel paths between *Guest* and *Hotel*. As before, it is required to investigate all paths separately but interestingly there is no final ambiguity in this scenario since only one of the two paths is abstractable.

Investigating this example in more detail, path (a) in Figure 10 can be abstracted since *Guest* inherits the unidirectional association from *Person* to *Account* (rule 3), *Hotel* has a bidirectional association to *Guest* (rule 97), and both of those associations can be abstracted to a simple, unidirectional association from *Hotel* to *Account* (rule 113). Path (b), on the other hand, is not abstractable because it is not possible to determine a calling dependency between *Hotel* and

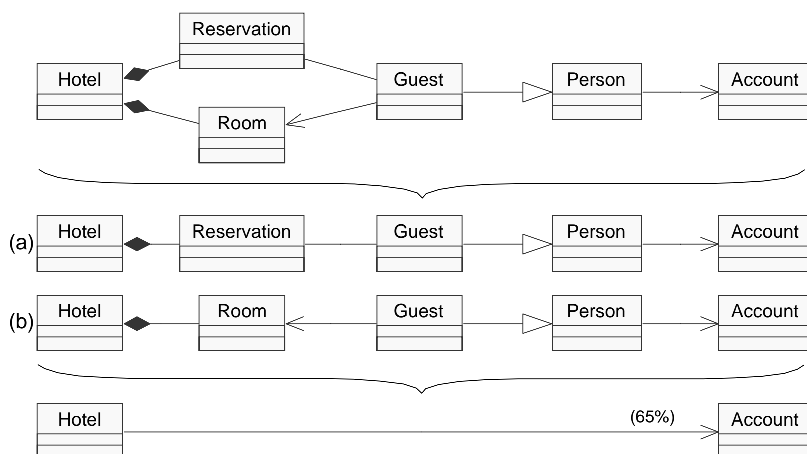

Fig. 10. Resolving ambiguities during the abstraction of parallel paths; Path (a) is abstractable and Path (b) is not abstractable which leads in nonambiguous abstraction result.

Account. This example shows that an original ambiguous premise (dual path) did not yield an ambiguous result.

5.5 Concurrent Abstraction of Multiple, Higher-Level Classes

Concurrent abstraction of multiple classes can significantly reduce the cost of serial abstraction by focusing on those paths only that are relevant. For concurrent abstraction, we define a set of classes that are important on a higher level. For instance, in context of Figure 2, the higher-level diagram defined the classes *Hotel*, *Guest*, *Payment*, and *Expense* as important; ignoring classes such as *Account*, *Reservation*, or *Person* (helper classes). If one would investigate all paths among *Hotel*, *Guest*, *Payment*, and *Expense* then one would find eight possibilities (Figure 11 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*.

Under normal circumstances, all eight paths would have to be abstracted to determine the abstract relationships among the four given important classes *Hotel*, *Guest*, *Payment*, and *Expense*. During concurrent abstraction, however, it is not desired to know about the abstract relationships between any two classes of the *Hotel-Guest-Payment-Expense* set; instead, it is desired to know the abstractions of all four classes together. Take, for instance, the path *Hotel-Reservation-Guest-Person-Account-Transaction-Expense* in Figure 11 (top). Abstracting this path reveals a unidirectional association from *Hotel* to *Expense*. The negative aspect of exploring this path is that it eliminates the class *Guest* because it is a helper class between *Hotel* and *Expense*. This is invalid here since one cannot declare *Guest* as 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

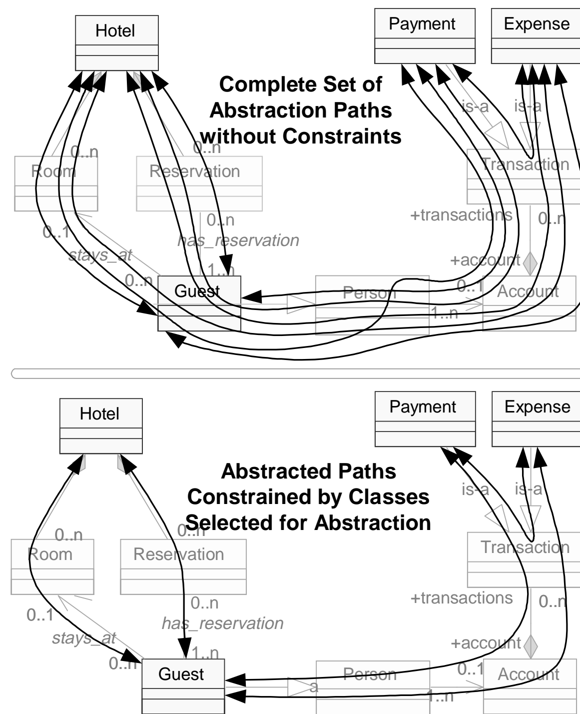

Fig. 11. Concurrent abstraction constrains abstraction paths (nine theoretical paths on the top versus four actual paths on the bottom).

former relationship (*Hotel-Expense*) is an abstraction of the latter two relationships. It follows that our abstraction approach only allows the abstraction of paths that purely consist of unimportant helper classes.

Figure 11 (bottom) depicts the subset of abstraction paths from Figure 11 (top) that do not contain any important classes. As can be seen, abstracting multiple, higher-level classes concurrently constrains the possible set of abstraction paths and also make abstraction computationally less expensive (see also Section 8).

5.6 Reuse during Abstraction

Earlier, it was shown that serial abstraction is exhaustive and can be computationally expensive since all abstraction combinations have to be investigated. Section 5.2 introduced "merging" as a way of coping with computational scalability. This section discusses "reuse" as another, complementary method to improve computational scalability. Abstraction has the benefit of extensive reuse of previously derived abstraction results. Consider Figure 12 (lower half), which recaptures the results of abstracting the *Guest-Person-Account-Transaction-Expense* pattern from Figure 8 (note that Figure 12 depicts results after merging). Reuse during abstraction makes use of previously generated abstraction results (intermediate and final ones) to simplify subsequent abstractions. Figure 12 (upper half) shows that abstracting the *Guest*-to-*Payment*

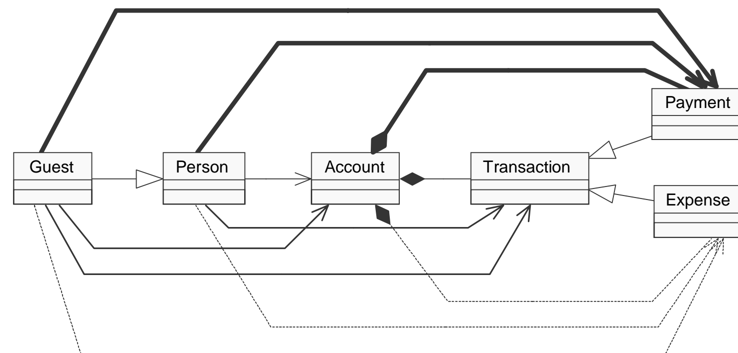

Fig. 12. Reuse during abstraction: Abstraction of *Guest*-to-*Expense* (bottom) path produces three intermediate abstraction results that can be reused during abstraction of *Guest*-to-*Payment* path (top).

pattern only leads to three new abstraction results since it can reuse three other (intermediate) abstraction results that were previously generated by abstracting the *Guest*-to-*Transaction* pattern. Section 8 will show metrics about the strong leverage of reuse in improving the computational complexity during abstraction.

5.7 Blind Alleys during Abstraction

Our abstraction approach does not need to investigate classes and relationships that are not part of a path (blind alleys). As such, there is no need to investigate *Hotel* or *Reservation* if it is only of interest to find the abstract relationship between *Guest* and *Expense*; or there is no need to investigate *Payment* if it is only of interest to find the abstract relationship between *Guest* and *Expense*. In the first case, the class *Hotel* and *Reservation* are not part of any path. In the second case, the class *Payment* is a detour of a valid path that terminates shortly thereafter. The rationale for ignoring blind alleys is to avoid meaningless circularities (e.g., *Transaction* would have to be visited twice if *Payment* should become part of the path from *Guest* to *Expense*).

5.8 Abstracting Cardinalities

Our abstraction rules emphasize the syntactic structure of "boxes" and "arrows" in class diagrams. However, class diagrams consist of more than just boxes and arrows. Figure 13 (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 is shown 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).

Figure 13 (right) shows an abstraction problem that was (partially) solved in Section 5.5. It was shown there that the class structure *Hotel*-*Room*-*Reservation*-*Guest* can be abstracted into two (ambiguous) relationships: a

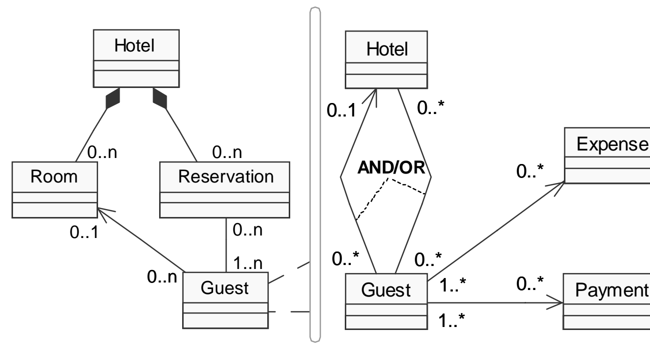

Fig. 13. Abstracting cardinalities.

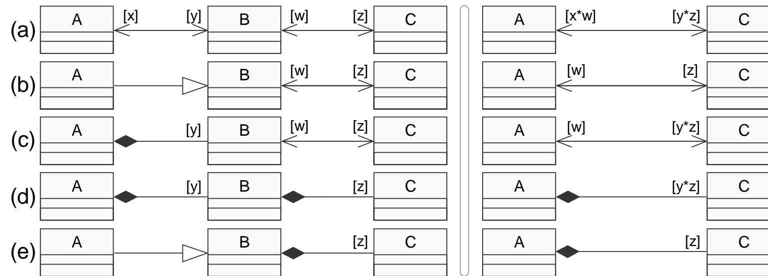

Fig. 14. Abstracting cardinalities of association and aggregation relationships.

bidirectional association and a unidirectional association (Figure 13 (right)). The following shows how cardinality information can be added to these two abstract relationships. The reasoning is as follows: If 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), then one can derive the transitive property that a *Guest* may stay at most one *Hotel* at any given point in time. 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. Figure 13 also shows other abstracted cardinalities among *Guest Expense* and *Payment*.

Figure 14 depicts common scenarios of abstracting cardinalities. Since only association and aggregation relationships can have cardinalities in UML, abstracting cardinalities is only needed for those rules in Table I that result in associations or aggregations. Scenario (a) in Figure 14 points out that the cardinalities of the left hand sides and the right hand sides have to be multiplied separately if two associations are abstracted. When cardinalities have ranges, the lower-bounds of the ranges have to be multiplied separately from the upper-bounds (e.g., [0..1]*[0..4] = [0..4]; [1..3]*[2..7] = [2..21]; [0..n]*[1..2] = [0..n]). In case, a cardinality is a single digit (i.e., [1]), it can be interpreted as a

range with both lower bound and upper bound equal the digit value (e.g., [1] ≡ [1..1]). Figure 14(b)–14(e) describe additional scenarios on how to abstract cardinalities of other relationships. Our approach enacts cardinality rules after abstraction candidates have been identified.

6. ABSTRACTING THE EXAMPLE PROBLEMS

Section 3 discussed the inadequacy of simple grouping for abstracting class structures and it motivated the need for a better abstraction technique. This section briefly summarizes how our abstraction technique improves on the problems discussed there.

Figure 13 (right) showed the result of abstracting the lower-level diagram in Figure 2 to make it analogous to the higher-level diagram presented there. Indeed, we find that our abstraction process produces comparable results although the simple grouping of classes would have failed to do the same (recall the discussion in Section 3.1).

Section 3.3 discussed the problem of grouping in context of the physical refinement in Figure 1. The main challenge was to understand how the subsystems *ServicePackage* and *FinancialPackage* relate to one another. We then pointed out why grouping failed to produce adequate results since generalization relationships cannot be transformed into "uses" relationships among packages (recall Figure 5). We define the following mapping among UML class associations and package relationships:

```
{1} Given: AssociationLeft ⇒ UsesLeft
{2} Given: AssociationRight ⇒ UsesRight
{3} Given: AssociationLeft[Agg] ⇒ UsesLeft
{4} Given: [Agg]AggregationRight ⇒ UsesRight
```

The above list shows that association and aggregation relationships can be mapped directly to "uses" relationships. Generalization relationships are ambiguous. They may or may not relate to "uses" relationships depending on their interactions with their neighbor elements (recall discussion in 3.1). In order to determinate the "uses" relationship between *FinancialPackage* and *ServicePackage*, we need to eliminate all generalization relationships and replace them with their semantic interpretations. In context of the HMS example, we need to abstract the class diagram in Figure 1 so that the generalizations disappear. Figure 15 shows the result of eliminating all generalizations. It can now be seen that *Guest* has an unidirectional association relationship to *Account* (`AssociationRight`) which, in turn, implies that *ServicePackage* uses *FinancialPackage*.

Finally, Section 3.2 defined several use cases (requirements) that needed to be ensured in the HMS model. In Figure 12, we showed how to abstract the lower-level diagram in such a manner that the direct relationships among *Guest*, *Payment*, and *Expense* become visible (Figure 4(a)). We can now use this abstraction to validate that "Guest may have Payment or Expense Transactions." We also validated "Guest may have zero or one Account" in Figure 15 where we showed the result of abstracting the relationship between *Guest* and

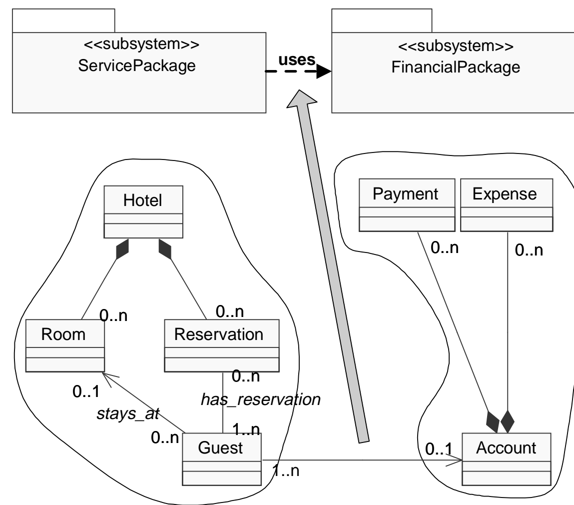

Fig. 15. Eliminating generalization relationships to determine "uses" relationships between *ServicePackage* and *FinancialPackage*.

```
abstractMultipleClasses(list of important classes):
1.  abstract_paths = ∅
2.  for all source ∈ list, target ∈ list, source ≠ target
3.     paths = find all paths between source and target
          - ignore paths that are circular or contain classes from list
4.     for all p ∈ paths
5.        abstract_results = abstractPath(p)
6.        abstract_paths insert abstract_results
7.  return abstract_paths
```

Fig. 16. Abstraction algorithm for concurrent abstraction.

Account (recall Figure 4(c)). There is no need for abstracting Figure 4(b) since that class pattern is on the same level of granularity as the lower-level diagram.

7. AUTOMATION AND TOOL SUPPORT

Our abstraction technique is fully automated to reduce manual effort and to make abstraction results reproducible. This section describes the abstraction algorithm and it presents its tool support called UML/Analyzer.

7.1 Algorithm

Figure 16 depicts the basic abstraction algorithm. As input, a list of important classes has to be provided. In context of the example in Figure 11, the list of important classes was *Hotel*, *Guest*, *Expense*, and *Payment*. The algorithm `abstractMultipleClasses` (Figure 16) first identifies possible abstraction paths followed by abstracting each path separately via the algorithm abstractPath (Figure 17). The following discusses both algorithms in detail.

```
abstractPath(path of relationships and classes)
1.  if (size of path = 0) return ∅ (empty list)
2.  if (size of path = 1) return new list containing path element
3.  if (abstraction for path exists) return list of all abstract elements
4.  if (exists rule-input-pattern(path))
       add relationship of type rule-result-pattern(path) to model
       return new list containing rule-result-pattern(path)
5.  abstracted_paths = ∅ (empty list)
6.  results_partial_path = abstractPath(sub path from element 0 to length-2)
7.  for all r ∈ results_partial_path
       if (exists rule(input) that matches {r + last two elements of path})
          abstracted_paths insert rule(result)
8.  results_partial_path = abstractPath(sub path from element 2 to length)
9.  for all r ∈ results_partial_path
       if (exists rule(input) that matches {first two elements of path + r})
          abstracted_paths insert rule(result)
10. for all p1 ∈ abstracted_paths
       if (exists p2 ∈ abstracted_paths where p1 is weaker or equivalent to p2)
          remove p1 from abstracted_paths
       else
          add relationship p1 to model
11. if (size of abstracted_paths > 1)
       mark elements in abstracted_paths as AND/OR ambiguity
12. return abstracted_paths
```

Fig. 17. Abstraction algorithm for abstracting single paths.

To support concurrent abstraction (recall Section 5.5), `abstractMultipleClasses` provides an interface for specifying a set of classes considered important from an abstract perspective. The algorithm for `abstractMultipleClasses` first defines a variable that will contain all abstraction results among the given set of classes (Step 1). Then it identifies all paths between all source and target classes (all combinations of the list of important classes). A path may not be circular (Section 5.7) nor may it contain important classes (Section 5.5). For all paths found, `abstractPath` is called to determine their abstract result(s). Note that a path is a sequence containing a relationship as the first and last elements plus classes and relationships in between (e.g., `AssociationLeft - Class - GeneralizationRight`)—analogous to our abstraction rules in Table I. The abstracted results are stored in a collection variable and returned to the caller.

The abstractPath algorithm determines the abstract interpretation(s) for a given path (e.g., path *Hotel-Reservation-Guest-Person-Account* in Figure 10). If the given path cannot be abstracted directly, it is broken up into smaller pieces and the algorithm then calls itself recursively until the path can be abstracted or it becomes too small. If the given path contains no element (Step 1), then an empty list is returned (no abstraction). If the given path contains at least one element (must be relationship), then that relationship is the abstraction (Step 2). If the path is more complex (3, 5, 7, 9... items) but an abstraction already exists, then the existing abstraction result is reused (Step 3). If no abstraction exists, it is attempted to find an abstraction rule from Table I that has an input pattern equals the given path (Step 4). If such a rule exists, then its result pattern is returned.

If no direct abstraction is possible then collection variable abstracted_paths is defined (Step 5). It will hold all abstract interpretations of the given path. Since no rule exists to abstract the given path, the path is broken up into smaller paths (one subset of the path excluding the first two items; the other excluding the last two items). Steps 6 and 8 then call abstractPath recursively upon which we can assume that a list is returned containing all found abstract relationships (i.e., the list will contain no value if no abstraction was possible; it will hold one value if the abstraction was unambiguous; otherwise it will hold multiple values). Steps 7 and 9 then take the returned abstractions, append or prepend the earlier removed elements and compare the resulting pattern against the rule set (Table I). Each identified abstraction result is then included in the collection variable abstracted_paths. In Step 10, results may be excluded if and only if another abstraction result exists that is considered equivalent (i.e., same type and directionality). The remaining elements in abstracted_paths are then added to the model and marked ambiguous if multiple, ambiguous interpretations exist (Step 11). In a final step, the elements in abstracted_paths are returned to the caller.

Note that abstractPath is recursive and ensures that all possible subsequences of a given path are abstracted (recall Section 5.2). If a path is too complex for direct abstraction, it is recursively broken down into smaller and smaller pieces until the abstraction results of the pieces can be determined. The return loop of the recursion then combines the abstraction results of the pieces to abstract them further. This guarantees that abstractPath returns a list containing only relationships that are determined to be the abstract interpretations of the given path. Our algorithm also "memorizes" abstract interpretations (by adding them to the model) so that abstraction results can be reused later (recall Section 5.6).

7.2 UML/Analyzer Tool Support

Our approach was co-developed with Rational Software [Egyed and Kruchten 1999] who developed a tool called Rose/Architect to support our abstraction technique (construction of Rose/Architect was subcontracted to Ensemble Systems by Rational Corporation). Since then, the approach was extended and the author developed a second, nonproprietary tool called UML/Analyzer. UML/Analyzer supports model transformation and consistency checking and also implements the abstraction technique discussed in this article. Figure 18 depicts some screen snapshots of the UML/Analyzer tool which is integrated with Rational Rose™ for the purpose of creating and modifying diagrams (synthesis). Rational Rose models can be downloaded, abstracted, and uploaded for visualization. Figure 18 shows Rational Rose™ in the lower windows and the UML/Analyzer main window to the upper left (depicting the repository view of the HMS system).

The tool uses abstraction rules (upper-right; Table I) to transform UML models. The class diagram in the lower left is an abstraction of the lower-level diagram in Figure 2 using *Guest*, *Hotel*, *Payment*, and *Transaction* as the important classes (recall Figure 11). The lower right window in Figure 18 shows

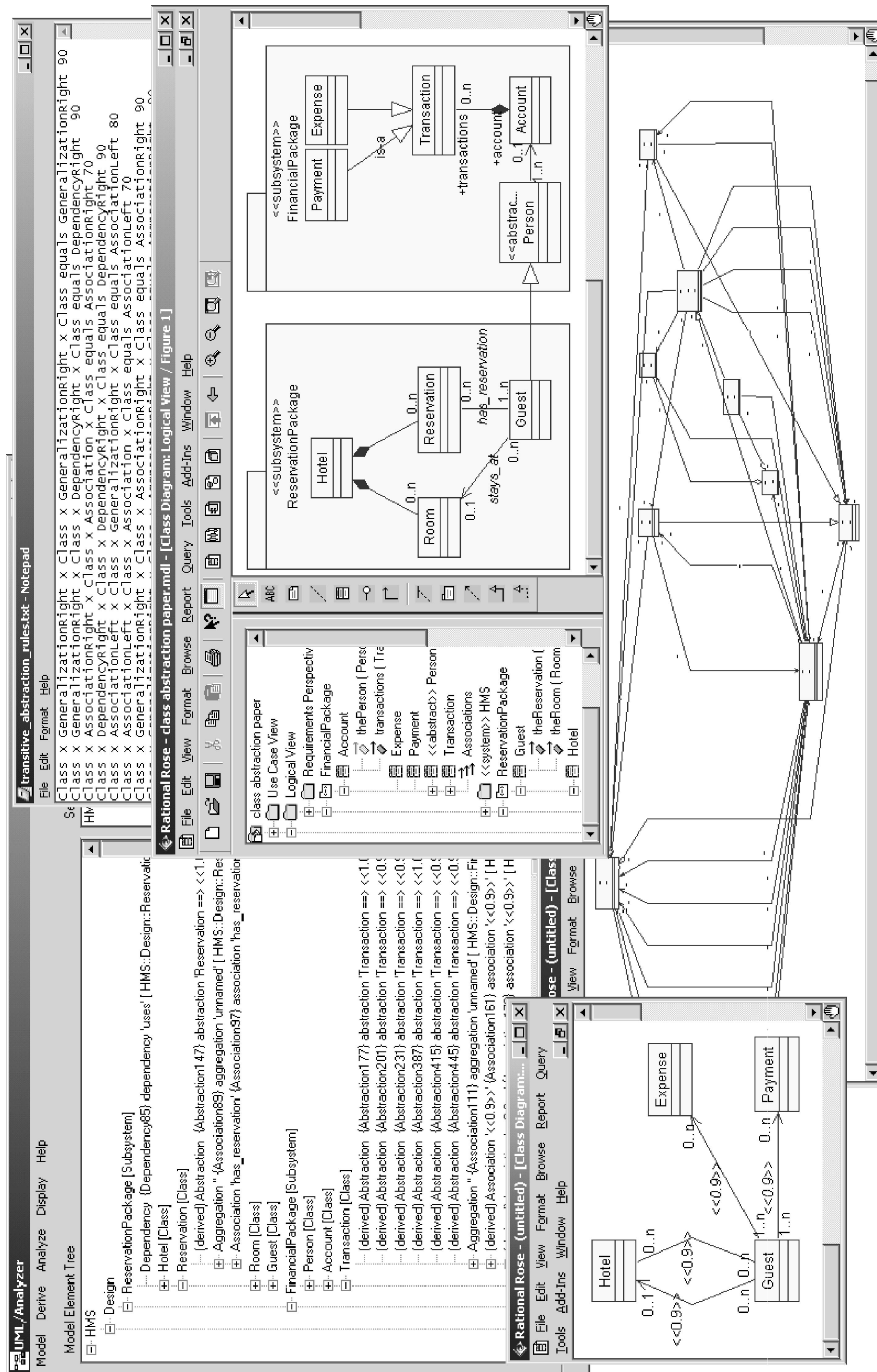

Fig. 18. UML/Analyzer tool supports class abstraction.

Table II. Models used for Validation

	Model Owner	Implementation Stage	Verification Method	Original Classes	Original Relations
Cargo Router (C2SADEL)	UCI	implemented*	model	29	48
Aeronautical World	Ensemble Systems	model only	inspection	36	35
ANTS Visualizer	Teknowledge Corporation	implemented	code	32	62
Boeing Avionics Open Experiment	Boeing	implemented	none	626	1354
Building Management System	Esprit Systems Consulting, Inc.	implemented*	inspection	26	11
Hospital System	USC	model only	model	39	44
Hotel Reservation System (paper)	USC	model only	inspection	9	9
Hotel Reservation System	USC	model only	model	50	98
Insurance Network Fees & Claims	Esprit Systems Consulting, Inc.	implemented*	inspection	136	42
Inter-Library Loan System	USC	implemented	code	26	43
Spacecraft Telemetrie Subsystem	Aerospace Corporation	implemented*	inspection	19	30
UML 1.3 Meta Model	OMG	implemented	code	135	256
Video-On-Demand System	Kim Dohyung	implemented	code	47	184
				1210	2216

memorized, intermediate abstraction results for that example. At the current state, the UML/Analyzer tool supports all aspects of class abstraction discussed in this article.

8. VALIDATION

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

8.1 Validity of Abstraction Rules and Algorithm

Table II lists a representative subset of 12 models that were used to validate the technique. Many of the models were built by third parties and are implemented systems today. The "implementation stage" column in Table II differentiates the models that were implemented and the ones that were not (model only). In some cases, we did not have access to the implemented system (indicated as "implemented*"). In other cases, we reverse engineered the model from the implemented system if no model existed (e.g., Video-On-Demand, ANTS Visualizer). The sizes of the models varied substantially, ranging from 10 classes to many hundreds of classes.

To validate the approach we applied different validation methods. In case the source code of the system was available (e.g., ANTS Visualizer), the abstraction results produced by the approach were validated against the actual source code. This method was applicable to four models and we consider it the most reliable validation method since the source code itself is the ground truth. In three cases, we validated the abstraction of our approach against existing higher-level abstractions. For instance, the Cargo Router was also modeled by a third-party using the C2SADEL architectural description language and we used the abstraction technique to translate the UML design into a class structure that was comparable to the architecture. We then validated our technique in its effectiveness to reproduce the architecture. Third, we used inspection in five cases where neither source code (see implemented*) was available nor

Table III. Excerpt of 18 Experiments Used to Validate our Abstraction Technique

	Abstract Classes	Abstract Relations	Transitive Relations Found	Direct Relations Foud	True Positives	False Positives	Ambiguities (AND/OR)	Percentage of Bad Results	Percentage of Ambiguities	Class Combinations (n*(n-1)/2)	Average Path Length	Average Path Breadth	Complexity Factor (APB^APL)	Investigated Paths
Cargo Router (C2SADEL) (1)	6	18	14	4	17	1	1	5.9%	5.9%	406	4.83	1.66	11.56	22
Cargo Router (C2SADEL) (2)	11	30	28	2	29	1	3	3.4%	10.3%	406	2.63	1.66	3.79	48
Aeronautical World	9	19	12	7	17	2	0	11.8%	0.0%	630	4.00	0.97	0.89	20
ANTS Visualizer (1)	11	15	9	6	15	0	0	0.0%	0.0%	496	2.91	1.93	6.78	64
ANTS Visualizer (2)	4	8	8	0	8	0	2	0.0%	25.0%	496	8.00	1.93	192.51	30
Building Management System	3	9	9	0	9	0	3	0.0%	33.3%	325	8.66	0.42	0.00	5
Hospital System (1)	14	14	11	3	14	0	0	0.0%	0.0%	741	2.79	1.13	1.41	31
Hospital System (2)	8	8	4	4	8	0	0	0.0%	0.0%	741	4.87	1.13	1.81	13
Hotel Reservation System (paper)	4	4	4	0	4	0	0	0.0%	0.0%	36	2.25	1.00	1.00	5
Hotel Reservation System (1)	14	36	28	8	35	0	10	0.0%	28.6%	1225	3.57	1.96	11.05	59
Hotel Reservation System (2)	4	3	3	0	3	0	0	0.0%	0.0%	1225	12.50	1.96	4499.88	8
Insurance Network Fees & Claims	14	35	20	15	35	0	5	0.0%	14.3%	9180	9.71	0.31	0.00	35
Inter-Library Loan System	6	14	14	0	10	4	2	40.0%	20.0%	325	4.33	1.65	8.74	424
Spacecraft Telemetrie Subsystem	6	28	25	3	26	2	2	7.7%	7.7%	171	3.16	1.57	4.16	37
UML 1.3 Meta Model	23	94	50	44	88	6	9	6.8%	10.2%	1378	2.30	1.84	4.07	104
Video-On-Demand System (1)	15	42	5	37	42	0	0	0.0%	0.0%	1081	3.13	3.91	71.37	57
Video-On-Demand System (2)	9	19	6	13	19	0	1	0.0%	5.3%	1081	5.22	3.91	1233.57	811
Video-On-Demand System (3)	9	22	8	14	22	0	5	0.0%	22.7%	1081	5.22	3.91	1233.57	601
	170	418	258	160	401	16	43	4.0%	10.7%	21024	4.17	1.66	5.47	2374

abstractions existed. Inspection was a manual process conducted by the author to investigate each abstraction result individually to reason about its correctness. The only model in Table II that was not validated was the Boeing Avionics Open Experiment model. It was not validated because of the author's insufficient familiarity with its domain, but it was included here because it is the largest model the technique was applied on which is relevant for computational scalability (see Section 8.3).

The given set of models and additional ones not presented in this article allowed us to validate our abstraction rules. It was discussed earlier that the UML semantics are not strong enough for full precision. For instance, if an association follows another association relationship, then it is unspecified whether the one really calls the other. This is a UML notation problem because, in a class diagram, one cannot specify how associations relate transitively. Our abstraction results must therefore be seen as indications of relationships but not factual ones. Given that there are often large numbers of potential transitive interdependencies among classes but only few factual ones, it may considerably weaken our approach if it indicates many of the wrong relationships (false positives). It was therefore our goal to validate the accuracy and correctness of our approach by measuring the number of false positives produced.

Table III shows an excerpt of 18 experiments conducted to validate our abstraction technique. The experiments were based on the 12 models in Table II where some models were abstracted in different ways depending on the viewpoints of their designer. In Table III, we show single abstraction scenarios for

most models, two different abstraction scenarios for four models (e.g., Cargo Router), and three abstraction scenarios for one model (Video-On-Demand). The choice of the number of experiments per model was random. The following discusses the columns of Table III:

—*Abstract Classes*. The number of important classes.
—*Abstract Relations*. The number of relationships among important classes generated by our approach (= transitive relations + direct relations).
—*Transitive Relations Found*. The number of abstract relationships that eliminated at least one helper class.
—*Direct Relations Found*. The number of abstract relationships that did not eliminate any helper classes.
—*True Positives*. The number of abstract relationships generated by our approach that were validated to be correct. The validation methods used were comparison with source code, comparison with existing models, and manual inspection (see Table III).
—*False Positives*. The number of abstract relationships generated by our approach that were validated to be incorrect.
—*Ambiguities (AND/OR)*. The number of ambiguous, abstract relationships generated by our approach.
—*Percentage of Bad Results*. Percentage of false positive versus true positives.
—*Percentage of Ambiguities*. Percentage of ambiguous abstraction results versus true positives.
—*Class Combinations*. The number of potential paths among important classes. It is computed through the formula $n^*(n - 1)/2$ where n is equal the number of abstract classes. Each class combination needs to be investigated to identify potential paths among classes and each such path leads to zero, one, or more abstract relationships.
—*Average Path Length*. Number of original classes divided by the number of abstract classes. This number indicates how many helper classes existed among important classes. This number is also referred to as abstraction ratio.
—*Average Path Breadth*. Number of original relationships divided by number of original classes. This number indicates the number of relationships of a class.
—*Complexity Factor*. It is computed through the formula (Average Path Breadth) raised to the power of (Average Path Length). The computational complexity of abstraction depends on the path length and the path breadth (see Section 8.3). Our approach requires the investigation of all paths and the number of paths is based on all combinations of helper classes among important classes. The complexity factor is an average indicator for this complexity.
—*Investigated Paths*. The number of paths among important classes identified by our approach. Each investigated paths is abstracted and may lead to zero or one abstract relationships.

In total, considering the large number of models, experiments, and model elements involved, we are confident in stating 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 case of the Inter-Library Loan system, 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 8.2 later about manual versus automated abstraction will discuss that it is significantly cheaper to manually inspect all abstraction results produced by our approach than it is to abstract manually.

Table III also shows that our approach only produced a small number of ambiguous results (recall Section 5.3 about AND/OR ambiguities). Our approach errs in favor of identifying ambiguities in case of doubt but, in context of the 18 experiments conducted, we found that 89% of the correct abstraction results were nonambiguous and only the remaining 11% could be merged.

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 who 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 who have no abstraction results (True Negatives)/(True Negatives+False Positives). True negatives were computed by subtracting abstracted paths from all investigated paths (see Table III).

In summary, our technique finds all abstract relationships and it does so with 99.3% specificity (low false positives). This finding is based on a validated set of 18 experiments on 12 different models of roughly 1500 classes and relationships. On a final note, it may seem surprising that only 10% of all investigated paths were abstractable although 70% of all abstraction patterns in Table I were abstractable. Since abstraction rules are applied in sequence and given that nonabstractable subpaths make whole paths nonabstractable, the simple answer is that abstraction is strongly dependent on path length and path breadth. The longer a path, the less likely a path is abstractable. The broader a path, the more likely a path is abstractable.

8.2 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 user does not get overwhelmed with too much (wrong) information and consequently incorrect abstraction results can be identified within reasonable effort. It can be seen in Table III 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 (ratio 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 to have 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 user from the extremely time consuming task of inferring abstract relationships among all class combinations and potential paths. Table III showed 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, Table III showed that there were 2374 different paths among all 170 abstract classes. Most of those paths were not abstractable and it follows that there was still a 10-fold benefit in manually validating our abstraction results versus manually abstracting those paths. This data shows 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 in 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 is generally straightforward and fast to judge the correctness of abstraction results.

Figure 19 shows the complexity involved in abstracting a class diagram of roughly 29 classes (the Cargo Router System). Even with all optimizations enabled, one can observe an enormous benefit in using our fully automated tool since having to abstract manually would require considerable human effort. Proper class abstraction requires the exploration of all possible path combinations followed by the application of proper class abstraction rules. Our tool reduces this task to mere seconds. Figure 19 is thus a motivation that semi-automated class abstraction (i.e., Racz and Koskimies [1999]) is infeasible for large-scale systems. Automation is thus desirable for both quantitative and qualitative reasons.

8.3 Complexity of Abstraction

The computational complexity of pattern matching is known to be very costly [Fahmy and Holt 2000]. Indeed, without optimizations the abstraction of a path (`abstractPath` in Figure 17) would be exponentially complex. Without optimization, `abstractPath` would generate $2^{(\text{length of path-3})}$ final abstraction results generating an additional (length of path-1)* $2^{(\text{length of path-3})}$ intermediate abstraction results (intermediate abstraction results are partial abstractions due to serialization; recall Section 5.1). For instance, a path with 11 classes

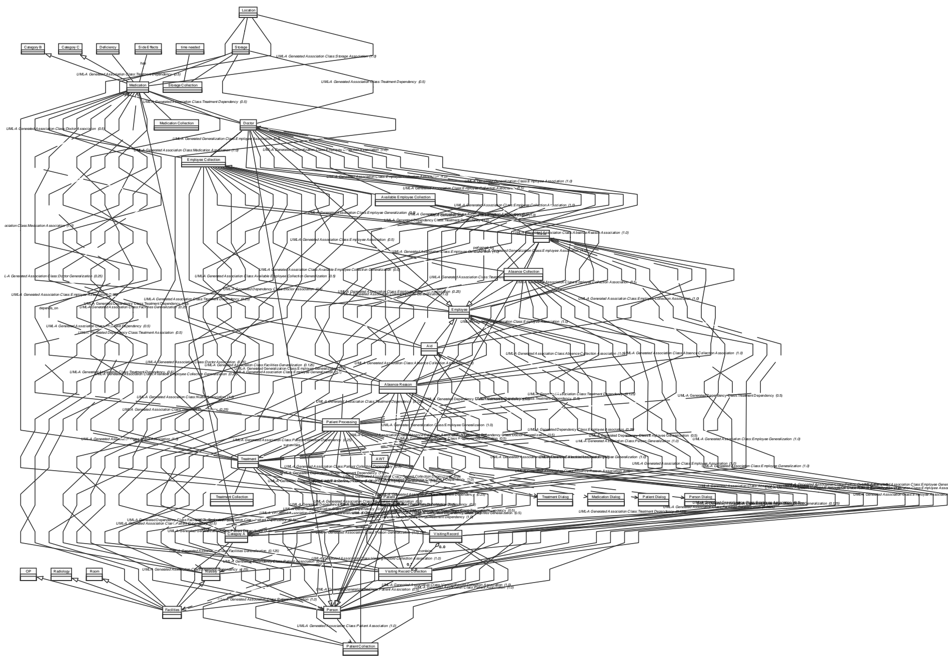

Fig. 19. Complexity of abstracting class diagrams motivates automation; Given example is the cargo router model with only 29 classes.

would abstract to 2067 intermediate and 256 final abstraction results. In order to reduce this complexity to a reasonable amount, intermediate abstraction results are reused (Section 5.6) and similar abstraction results within the same (sub) path are merged (Section 5.2). These optimizations reduce the computational complexity of path abstraction to polynomial time.

Reuse is mostly important during concurrent abstraction but it also supports `abstractPath` since the abstraction of a single path may explore partially overlapping subpaths (recall Section 5.2). In experiments, reuse has reduced the computational complexity by two-thirds (e.g., a path with 11 classes would abstract to 776 intermediate and 256 final abstraction results). Merging produces the most leverage because it combines abstraction results if they are of the same type, directionality, and connecting to the same classes (recall Section 5.2). Merging makes path abstraction polynomial complex—$O(n^2)$. As such, a path with 11 classes abstracts to 44 intermediate and 1 final abstraction result, assuming no ambiguity is encountered along the way. AND/OR ambiguities during the abstraction of a single path are rare in our approach but since the designer has the option of altering abstraction rules, they may occur. Each introduced ambiguity during the abstraction of a path may potentially double the number of intermediate and final abstraction results.

Figure 20 depicts the computational complexity during path abstraction for variable path length assuming no ambiguity. As can be seen, the simple nature of our abstraction patterns combined with reuse and merging results in very fast abstraction (note the logarithmic scale on the "y"-axis).

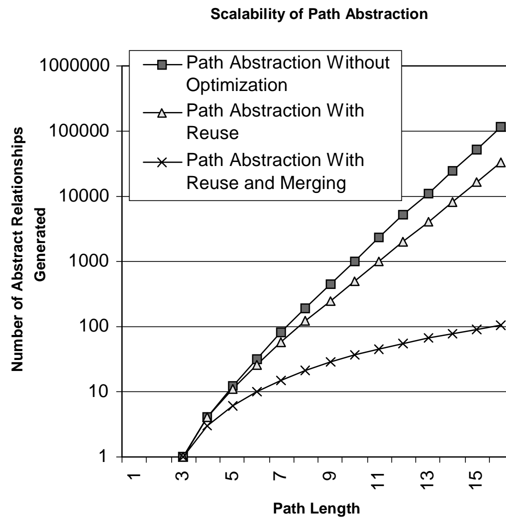

Fig. 20. Improving computational complexity.

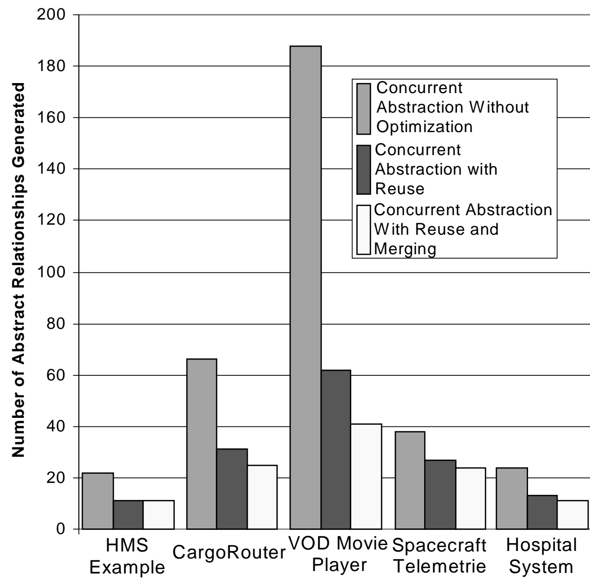

Fig. 21. Improving computational complexity and quality of results during concurrent abstraction.

Besides improving the computational complexity of `abstractPath`, we also improved the complexity of `abstractMultipleClasses`, which calls `abstractPath` for every path it identifies. Again, we reuse and merge results if they share overlapping paths and these improvements reduce the number of generated, abstract relationships by a factor of 2–5. Figure 21 visualizes these optimizations in context of five different class models. As can be seen, the most

significant optimization for `abstractMultipleClasses` is reuse since concurrent abstraction can make use of existing abstractions of previous paths. Merging also reduces the number of generated relationships.

It must be noted that reuse in `abstractMultipleClasses` saves computational time by not having to re-evaluate the same path twice. Merging, on the other hand, still requires the full evaluation (or reuse) of a path, but it improves the quality of results by eliminating redundant and overlapping information. In the context of `abstractPath`, both merging and reuse improve computational complexity.

Although our approach abstracts paths quickly and is thus significantly better than most existing abstraction approaches that rely on patterns (e.g., Fahmy–Holt), computational scalability may still cause problems. We observed that the number of paths may increase sharply with two primary factors contributing to the problem: (1) path length and (2) path breadth. The more helper classes per important class (average path length) and the more path alternatives (average path breadth), the more paths need to be investigated. The complexity factor in Table III showed that this factor increases fast since it is exponential (complexity = average path breadth raised to the power of average path length). As a consequence, it is our observation that the computational complexity of abstraction is not so much dictated by the number of model elements (the size of the model) but the "ratio of abstraction." The larger the ratio between important classes and helper classes during abstraction, the more computationally intensive is its abstraction as it increases the average path length and average path depth. This data, and our qualitative analysis about the correctness of abstraction results, implies that large-scale abstraction should be attempted in multiple, small, sequential steps.

9. DISCUSSION

This section discusses some border issues of class abstraction that are relevant for the usefulness of our approach:

9.1 Ambiguous Results

Our approach is based on heuristics to reason in the presence of ambiguous model definitions. We found that it is impossible to create "precise" abstraction rules in context of UML and many other modeling definitions. Consequently, we had the dilemma of how to deal with false positive or true negatives (both undesirable). We decided to err on the side of false positives (showing abstractions although there are none) since we found it easier to eliminate wrong abstraction results rather than generate missing abstractions. Our experience shows that our approach produces correct results in most cases and, as Jackson–Rinard [2000] argued, "even when incorrect, may provide a useful starting point for further investigation."

9.2 Extensibility of Rules

In Table I, we defined abstraction rules for four types of relationships. Our rules can be altered or extended to capture other types of relationships or even

classes. For instance, new rules could be created to support utility classes or parameterized classes. It is also possible to make use of UML extensibility mechanisms such as stereotypes .[Booch et al. 1999] to define model elements more precisely. We went through several iterations of refinement of our abstraction rules and found that the precision of our approach increases with the number of abstraction rules. It must be noted that our tool support for extensible rules is limited. Future work is needed to address this issue.

9.3 Methods and Attributes

Another problem is on how to abstract methods and attributes in classes. This article only described how class patterns can be abstracted into new, abstract relationships with directionalities and cardinalities. Currently, we do not abstract methods and attributes of classes. We will investigate this problem in future work.

9.4 Complex Abstraction Rules

Our abstraction rules are very simple and, in some cases, this simplicity may add to the ambiguity of the solution. It is possible to create more complex abstraction rules to counter this problem. Take, for instance, the following rule consisting of two helper classes and three relationships:

```
GeneralizationRight - Class - Association[Agg] - Class -
  GeneralizationLeft ⇒ Association[Agg]
```

Our approach supports new rules like the one above, but our approach is restricted to paths only. Thus, we do not support the creation of a single abstraction rule with blind alleys (Section 5.7) or parallel paths (Section 5.3). We are currently investigating how to incorporate a more extensible rule structure into our abstraction process. It is our hope that such extensibility leads to a richer rule set (i.e., representing design patterns [Gamma et al. 1994] as rules) without sacrificing computational scalability.

9.5 Manual Intervention

Although our abstraction technique is fully automated, we do find it necessary to allow human users to alter abstraction results or even perform abstraction semi-automatically. Our tool currently does not support human intervention during abstraction but the work of Racz and Koskimies [1999] shows how our tool could be extended to do so. The issue of manual intervention has a drawback in evolutionary terms. For instance, if models change, then how can one preserve abstraction results that were provided manually? We assume that considerable effort was spend creating them and thus they should not get lost. We are currently investigating the issue of "smart evolution" that deletes abstraction results only if they have become obsolete.

10. RELATED WORK

Many techniques have been proposed to aid the understanding of complex class structures. There are reading techniques such as inspection [Fagan 1986] 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 [Tarr et al. 1999]. Class structures can be subdivided into multiple views [Altmann et al. 1988; Finkelstein et al. 1991; Garlan 1988] where partial and potentially overlapping portions of the structure are depicted. The sum of all views (diagrams) is the complete class structure 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 structure.

Lieberherr et al. [1994] 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 article. 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 [2000] 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 rewriting 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 the problem of race conditions (i.e., preference to individual abstraction rules—Section 5.1) and ambiguities during abstraction (Section 5.3). Furthermore, their transformation algorithm is computationally very expensive since they can define complex patterns and antipatterns. Instead, our approach relies on relatively simple patterns that can be abstracted quickly (recall Section 8).

The works of Schuerr et al. [1995] 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 an heuristic-based approach that optimizes the use of a limited set of graph rewrite rules to achieve faster performance. The limitation of their improvement 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 allow a very simple rule pattern structure (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.

Snelting and Tip [1998] 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 structured class hierarchy. Although their work reinterprets 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 [1999] 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 serial and concurrent abstraction, abstraction reuse, and ambiguity handling. As a result, they did not devise an automatable abstraction technique but instead developed a tool for semi-automated use. In Section 8, we pointed out the disadvantages of semi-automated abstraction on largescale class diagrams. Irrespective of the drawbacks of their approach, we see their work as an 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.

Murphy et al. [1995] devised a reflexion model on how to abstract and relate classes in a complex class structure. They essentially group implementation classes into clusters and then observe method calls among those clusters dynamically (i.e., while testing scenarios). As a response, they can determine how clusters interacted with one another. Their technique has the disadvantage that observing the runtime behavior of clusters only gives information about calling dependencies, which is just one of the properties we are interested in. They cannot distinguish inheritance, dependency, or aggregation on an abstract level. Furthermore, their technique requires exhaustive dynamic analyses to produce abstractions. This is a very time-consuming effort if done manually.

Our abstraction technique is conceptually related to transformation techniques such as Sequence to Statechart transformation [Koskimies et al. 1998; Schönberger et al. 2001], Collaboration to Statechart transformation [Khriss et al. 1998], and Sequence to Class transformation [Tsiolakis and Ehrig 2000]. All these approaches recognized the fact that model transformation in general can be done without the use of intermediate, third-party languages. For instance, Koskimies et al. [1998] 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 to generate 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.

11. CONCLUSION

This article presented an approach for automated abstraction of class diagrams. Our approach exploits the semantic meaning of patterns of classes and their relationships to infer transitive properties. Although our abstraction rules are primitive in structure, they are rich enough in number to abstract large-scale class diagrams. To date, we have validated our abstraction technique and its rules on numerous third-party applications and models with up to several hundred model elements. We showed that our technique scales, produces correct results most of the time, and addresses issues such as model ambiguities that are inherently part of many (UML) diagrams. We demonstrated various forms of ambiguities and showed that there are ways of living with them—even preserving them during transformation.

Our abstraction process is fully supported through the public UML/Analyzer tool and partially supported through the proprietary Rose/Architect tool owned by Rational Corporation. UML/Analyzer provides a superset of the functionality of Rose/Architect, which only supports simple abstraction patterns (Section 4) combined with nonexhaustive path abstraction. Both tools are integrated with Rational Rose and both tools only require a low-level UML class diagram as well as a specification of what classes are important for abstraction. In the case of model understanding and reverse engineering, the list of important classes likely has to be provided by a human user. In the case of consistency checking, the list can also be derived automatically in some cases.

We find our abstraction technique to be well suited for model understanding, reverse engineering, and consistency checking. During model understanding, our technique provides users a lightweight, 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. Those interpretations can then be saved as abstract class diagrams. And for consistency checking, our approach provides a comparison infrastructure where a lower-level diagram can be abstracted so that the existing higher-level diagram can be compared with the abstracted lower-level diagram.

Future work is to investigate the applicability of our approach on other types of models. We will also investigate means of qualifying the likelihood of correctness of individual abstraction rules using reliability numbers. Those reliability numbers could then be used for approximating the correctness of abstraction results produced by our approach.

ACKNOWLEDGMENTS

We wish to thank Philippe Kruchten for the initial idea and support. We also wish to thank Barry Boehm, Cristina Gacek, Paul Grünbacher, Nenad Medvidovic, Dave Wile, and the anonymous reviewers for insightful discussions.

REFERENCES

ABI-ANTOUN, M., HO, J., AND KWAN, J. 1999. Inter-library loan management system: Revised life-cycle architecture. Center for Software Engineering, University of Southern California, Los Angeles, Calif.

ALTMANN, R. A., HAWKE, A. N., AND MARLIN, C. D. 1988. An integrated programming environment based on multiple concurrent views. *Austral. Comput. J. 20*, 2, 65–72.

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

BOOCH, G., RUMBAUGH, J. AND JACOBSON, I. 1999. *The Unified Modeling Language User Guide*. Addison-Wesley, Reading, Mass.

DOHYUNG, K. 1999. Java MPEG Player. http://mirage.snu.ac.kr/dhkim/java/MPEG/.

EGYED, A. 2000. Heterogeneous view integration and its automation. Ph.D. Dissertation, Univ. Southern California, Los Angeles, Calif.

EGYED, A. 2001. A Scenario-driven approach to traceability. In *Proceedings of the 23rd International Conference on Software Engineering (ICSE)*, 123–132.

EGYED, A. AND KRUCHTEN, P. 1999. Rose/Architect: A tool to visualize architecture. In *Proceedings of the 32nd Hawaii International Conference on System Sciences (HICSS)*, Maui, HI.

EGYED, A. AND MEDVIDOVIC, N. 2000. A formal approach to heterogeneous software modeling. In *Proceedings of 3rd Foundational Aspects of Software Engineering (FASE)*, Berlin, Germany. 178–192.

EGYED, A. AND WILE, D. 2001. Statechart simulator for modeling architectural dynamics. In *Proceedings of the 2nd Working International Conference on Software Architecture (WICSA)*, Amsterdam, The Netherlands. 87–96.

EVANS, A., FRANCE, R., LANO, K., AND RUMPE, B. 1998. The UML as a formal modeling language. *J. Comput. Stand. Interf. 19*, 325–334.

FAGAN, M. E. 1986. Advances in software inspections. *IEEE Trans. Softw. Eng. (TSE) 12*, 7, 744–751.

FAHMY, H. AND HOLT, R. C. 2000. Using graph rewriting to specify software architectural transformations. In *Proceedings of the 15th IEEE International Conference on Automated Software Engineering (ASE)*, 187–196.

FINKELSTEIN, A., KRAMER, J., NUSEIBEH, B., FINKELSTEIN, L., AND GOEDICKE, M. 1991. Viewpoints: A framework for integrating multiple perspectives in system development. *Int. J. Softw. Eng. Knowl. Eng.* 31–58.

GAMMA, E., HELM, R., JOHNSON, R., AND VLISSIDES, J. 1994. *Design Patterns Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, Mass.

GARLAN, D. 1988. Views for tools in integrated environments. In *Advanced Programming Environments,* 314–343.

GOTEL, O. C. Z. AND FINKELSTEIN, A. C. W. 1994. An analysis of the requirements traceability problem. In *Proceedings of the 1st International Conference on Requirements Engineering*, 94–101.

JACKSON, D. AND RINARD, M. 2000. Software analysis: A roadmap. In *Proceedings of the 20th International Conference on Software Engineering (ICSE)*, 133–145.

KHRISS, I., ELKOUTBI, M., AND KELLER, R. 1998. Automating the synthesis of UML statechart diagrams from multiple collaboration diagrams. In *Proceedings for the Conference of the Unified Modeling Language,* 132–147.

KOSKIMIES, K., SYSTÄ, T., TUOMI, J., AND MÄNNISTÖ, T. 1998. Automated support for modeling OO software. *IEEE Softw.* 87–94.

LIEBERHERR, K. J., HURSCH, W. L., AND XIAO, C. 1994. Object-extending class transformations. *J. Form. Asp. Comput. 6*, 4, 391–416.

MCUMBER, W. E. AND CHENG, B. H. C. 2001. A general framework for formalizing UML with formal languages. In *Proceedings of the 23rd International Conference on Software Engineering (ICSE)*, 433–442.

MUELLER, H., JAHNKE, J. H., SMITH, D. B., STOREY, M. A., TILLEY , S. R., AND WONG, K. 2000. Reverse engineering: A roadmap. In *Proceedings of the Conference on the Future of Software Engineering*, 47–60.

MURPHY, G. C., NOTKIN, D., AND SULLIVAN, K. 1995. Software reflexion models: bridging the gap between source and high-level models. In *Proceedings of the 3rd ACM SIGSOFT Symposium on the Foundations of Software Engineering.* ACM, New York, 18–28.

ÖVERGAARD, G. 1998. A formal approach to relationships in the unified modeling language. In *Proceedings of the Workshop on Precise Semantics for Software Modeling Techniques (PSMT'98)*, 91–108.

RACZ, F. D. AND KOSKIMIES, K. 1999. Tool-supported compression of UML class diagrams. In *Proceedings of the 2nd International Conference on the Unified Modeling Language (UML)*, 172–187.

SCHUERR, A., WINTER, A. J., AND ZUENDORF, A. 1995. Graph grammar engineering with PROGRES. In *Proceedings of the 5th European Software Engineering Conference (ESEC)*, 219–234.

SCHÖNBERGER, S., KELLER, R. K., AND KHRISS, I. 2001. Algorithmic support for model transformation in object-oriented software development. *Concur. Computat. Prac. Exp. 13*, 5, 351–383.

SIEGFRIED, S. 1996. *Understanding Object-Oriented Software Engineering*. IEEE Computer Society Press, Los Alamitos, Calif.

SNELTING, G. AND TIP, F. 1998. Reengineering class hierarchies using concept analysis. In *Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering.* ACM, New York, 99–110.

TARR, P., OSHER, H., HARRISON, W., AND SUTTON, S. M. JR. 1999. *N* degrees of separation: Multidimensional separation of concerns. In *Proceedings of the 21st International Conference on Software Engineering (ICSE 21)* (Los Angeles, Calif.). 107–119.

TSIOLAKIS, A. AND EHRIG, H. 2000. Consistency analysis of UML class and sequence diagrams using attributed graph grammars. In *Proceedings of GRATRA 2000* (Berlin, Germany). 77–86.

Received October 2001; revised April 2002; accepted September 2002

CORRECTION! On page 488, the paper incorrectly describes an article from Murphy et al. [1995]. Their technique allows a developer to define a high-level model of what the system is expected to be doing which is then compared to what the system is actually doing. While the high-level model is defined by a developer, the observation of what the low-level model is doing can be derived statically from source code or dynamically from the system's execution. The latter was misrepresented in this article. Their technique can be used to map (almost all) binary relationships one can extract between source-level program elements. Some common examples of relationships they look at are calls between functions (methods), accesses to global variables, event interactions, and inheritance. So contrary to our characterization, their technique is not limited to runtime behavior.