An Efficient and Scalable Approach to Correct Class Model Refinement

Wuwei Shen, *Member*, *IEEE*, Kun Wang, and Alexander Egyed, *Member*, *IEEE*

Abstract—Today, programmers benefit immensely from Integrated Development Environments (IDEs), where errors are highlighted within seconds of their introduction. Yet, designers rarely benefit from such an instant feedback in modeling tools. This paper focuses on the refinement of UML-style class models with instant feedback on correctness. Following the Model-Driven Architecture (MDA) paradigm, we strongly believe in the benefit of maintaining high-level and low-level models separately to 1) document the lower level model and 2) continuously ensure the correctness of the low-level model during later evolution (i.e., high- or low-level models may be evolved independently). However, currently the refinement and subsequent evolution lack automated support, let alone an instant feedback on their correctness (i.e., consistency). Traditional approaches to consistency checking fail here because of the computational cost of comparing class models. Our proposed instant approach first transforms the low-level model into an intermediate model that is then easier comparable with the high-level model. The key to computational scalability is the separation of transformation and comparison so that each can react optimally to changes—changes that could happen concurrently in both the high- and low-level class models. We evaluate our approach on eight third-party design models. The empirical data show that the separation of transformation and comparison results in a 6 to 11-fold performance gain and a ninefold reduction in producing irrelevant feedback. While this work emphasizes the refinement of class models, we do believe that the concepts are more generally applicable to other kinds of modeling languages, where transformation and subsequent comparison are computationally expensive.

Index Terms—Class models, consistency checking, refinement, separation of concerns, and UML.

1 Introduction

Class models, an integral part of the Unified Modeling Language (UML) [1], describe the structure of a software system and do so very effectively at any level of abstraction. During the early phases of software development, class models are used to capture a software system from a high-level perspective, showing the most significant components. Later on, lower level details are added by refining high-level classes and their relationships [2]. However, a low-level class model is only a correct refinement of a high-level class model if the added details do not change its meaning [3]. This is especially then desirable when high-level models have desirable properties (i.e., were validated against/for deadlock, performance, security) and engineers like to ensure that the lower level models correctly implement these properties; or if high-level models are retained in addition to lower level models for documentation and ease of understanding. Following the MDA paradigm, we, therefore, strongly believe in the need to maintain high-level and low-level models separately (i.e., to not discard the high-level model after refinement) to 1) document the lower level model and 2) continuously ensure the correctness of the lower level models even after the initial refinement. Unfortunately, when high- and low-level class models are evolved separately, they can easily get out of sync—we speak of inconsistencies [4]. This work is, thus, about helping engineers understand how changes in class models affect the correctness of their abstractions/refinements—irrespective of whether the high-level model was created first or not (top-down or bottom-up engineering).

The lack of automated support implies that such inconsistencies are often discovered later in the software life cycle (if at all) when they become increasingly costly to fix [5]. This is in contrast to a new trend in many modern Integrated Development Environments (IDEs), which inform programmers of errors within seconds of introducing them. Programmers benefit tremendously from such an instant feedback and we believe that designers would likewise benefit from such an instant design feedback—in our case, to continuously ensure the correct refinement of class models, if so desired. It is the engineers' decision when and how to resolve inconsistencies (i.e., tolerating inconsistencies [6], [7]). However, tolerating inconsistencies does not imply ignoring them—hence the need for quick, automated feedback in a nonintrusive manner.

Literature provides many approaches for identifying inconsistencies in design models [8], [9], [10]. These approaches usually require consistency rules (formal constraints) against which the design models are evaluated. In [11], we demonstrated that consistency checking can be implemented efficiently for many kinds of consistency rules. We also implemented such a consistency checking approach, called Integrated Abstraction and Comparison

- *W. Shen is with the Department of Computer Science, Western Michigan University, Kalamazoo, MI 49008. E-mail: wuwei.shen@wmich.edu.*
- *K. Wang is with Siemens PLM Software, 2600 Green Rd, Ann Arbor, MI 48105. E-mail: wangkun@siemens.com.*
- *A. Egyed is with the Institute for Systems Engineering and Automation (SEA), Johannes Kepler University, Linz, Austria. E-mail: alexander.egyed@jku.at.*

Manuscript received 25 Feb. 2008; revised 30 Jan. 2009; accepted 24 Feb. 2009; published online 1 Apr. 2009.
Recommended for acceptance by H. Ossher.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSE-2008-02-0085.
Digital Object Identifier no. 10.1109/TSE.2009.26.

(IAC), for ensuring the consistency among high-level and low-level class models. This approach, based on a known class abstraction technique [12], [13], embodies a large number of abstraction productions, where each abstraction production replaces two relationships and a class (e.g., A calls B and B calls C) with a transitive relationship (e.g., A calls C). The consistency checker benefits from these abstraction productions in that the rules first simplify the low-level class model (usually through multiple abstraction productions applied in sequence) and then compare the final abstraction result with the high-level class model.

However, we found that the IAC approach had severe scalability problems. These problems puzzled us and, while investigating it, we realized that most of the abstraction work performed was unnecessary. That is, different consistency rules reevaluated the same intermediate abstraction productions—a duplication that could not easily be avoided because of the independent nature of consistency rules. The dilemma of the IAC approach was that it combined two activities that did not fit well together: 1) the abstraction of the low-level model and 2) its comparison with the high-level model. Since our consistency rules were written such that comparison dictated the use of abstraction productions, we were no longer able to distinguish between them. It should be noted that this performance problem was not the result of our inability to "tweak" the consistency rules to react differently to certain kinds of changes. Such tweaking would have been possible and would have resulted in better performance; however, it would have been complicated, error prone, and it would not have eliminated the real problem.

We, therefore, searched for a new way for dealing with the correct refinement of class structures—a way that separated the use of abstraction from comparison [14]; and we developed a second tool, called Separated Abstraction and Comparison (SAC). In SAC, both abstraction and comparison had their own rules and were invoked separately in ways that were ideal for them individually without considering the needs of the other (separation of concerns). Only an ordering was imposed in that abstraction always had to happen before comparison such that comparison could trust the correctness of the abstraction results. The SAC approach, thus, remembered and made use of previous abstraction results (through an intermediate class model) and consequently avoided unnecessary abstractions.

This paper demonstrates that this separation of concerns is highly beneficial for the consistency checking of software models as follows:

1. We observed a 6 to 11-fold performance improvement over the IAC approach evaluated on the same third-party models.
2. The SAC rules were much easier to write and evaluate due to the separation of concerns.
3. SAC was more memory efficient.
4. SAC produced less irrelevant feedback because it abstracted and compared significantly less than IAC. Of course, SAC was as reliable as IAC in terms of correctness and completeness.

This paper, thus, contributes an approach to the consistency checking among different levels of class models that is significantly better than that of traditional consistency checking. While this paper focuses on the correct refinement of class models, our approach may equally be applicable in other situations, where consistency checking is expensive. Our approach could also be used to sync class models and source code since the source code could be represented as a low-level class model.

This paper is organized as follows: Section 2 discusses related work. Section 3 presents the definition of consistency followed by some related information about SAC/IAC. We fully discuss the IAC and SAC approaches in Section 4. Section 5 evaluates both IAC and SAC in terms of time, memory cost, and accuracy. We draw a final conclusion in Section 6. Note that this work is based on a previously published abstraction technique [12], [13], [15]. The incremental implementation of this abstraction technique, as used in SAC, was published in [16]. Neither IAC nor SAC were published previously although IAC is conceptually alike traditional approaches to consistency checking [10], [11], [17]. A batch version of IAC appeared in [14], though it has little resemblance with IAC because much of the complexity of IAC is about its ability to react to model changes.

2 Related Work

Various approaches aimed at finding inconsistencies in UML models have been presented in the literature. We will discuss these related approaches by means of three criteria.

The first criterion is the distinction between vertical and horizontal consistency checking [18]. Vertical consistency checking is meant to compare models at different levels of abstraction. Horizontal consistency checking is meant to compare various models at the same level of abstraction. Existing work for vertical consistency checking usually supports some refinement relationship as is discussed in this paper. There exist some approaches, which directly support class structure refinement. Lieberherr et al. [19] define a set of transformations, which captures class evolution. These transformations can then be applied to find inconsistencies between two class models at different levels of abstraction. Whittle [20] proposes a set of transformation rules investigating the consistency relationship between two class models at different levels. Shen and Low [21] discuss how to apply a profile mechanism to represent refinement rules based on different levels of class models. However, our approach goes beyond the batch abstraction of entire class structures and focuses on incremental abstraction and comparison to ensure instant feedback on correctness during refinement. Not restricted to class models is the approach by Berenbach [22]. There, models designed at different phases, such as analysis and design, are compared. Berenbach also presents heuristics and processes to create verifiable UML analysis and design models. Based on a set of heuristics, various analysis and design models can be compared to find inconsistencies.

Most existing work on consistency checking, however, focuses on horizontal consistency checking, which ensures the consistency among different models at the same level of abstraction (typically class, sequence, or statechart models). Tsiolakis and Ehrig [23] propose a method for finding inconsistencies among UML class models, use case diagrams, sequence diagrams, and statechart diagrams. Engels et al. [24] detect inconsistencies between a class diagram and statechart diagram. Straeten et al. [25] investigate inconsistencies among UML class diagrams, sequence diagrams, and statechart diagrams. These horizontal consistency checking approaches, however, benefit from much simpler consistency rules, where the role of transformation is limited to data gathering and simple manipulation. Horizontal consistency checking can, thus, be expressed in numerous albeit simple consistency rules.

How the consistency checking approaches are implemented is the second criterion to divide the existing approaches. Many approaches rely on formal methods [26] as a platform for consistency checking. The most widely used formal methods include CSP [27], B [28], Z [29], and Description Logic [30]. The advantage of these formal methods is that they provide a precise semantics to UML diagrams. Consistency rules can thus be defined precisely. Rasch and Wehrheim [31] consider the inconsistencies between a class and its associated state machine. They combine Object-Z and CSP to describe the static and dynamic aspects of a software system and inconsistencies can then be found by the FDR model checker [32]. Similar to Rasch and Wehrheim's work, Yeung [33] also considers the inconsistency problem between a UML class and its state machine. However, Yeung applies CSP and B to formalize the static and dynamic aspects of a UML model. Finally, Treharne and Schneider's coupling [34] between CSP and B is employed to find inconsistencies. Finkelstein et al. [35] apply graph rewriting to translate statechart and sequence/activity diagrams into Petri Nets to find inconsistencies. Nentwich et al. [10] implement a tool called xLinkIt that detects inconsistencies among multiple, distributed XML models. Yao and Shatz [36] apply an Extended Colored Petri Net to check inconsistencies in the dynamic aspect of a UML model. Wagner et al. [37] apply graph grammars to find inconsistencies in UML models. There, a graph rewriting system searches for inconsistency patterns; and if a match is found, then an inconsistency is reported.

In some way, our approach is similar to these approaches because they separate transformation from comparison. That is, they transform the UML model into a different, usually formal, language to simplify the comparison there because the transformation unifies the language(s) used in the different models. However, our approach does not use transformation for the sake of unifying class models. Also, our goal is not to provide a better formal foundation for comparison but rather to separate transformation from comparison to optimize them individually.

How the feedback on inconsistencies is produced is another criterion to distinguish the current approaches. Almost all existing approaches implement consistency checking in a batch way. This is very inefficient because most model changes are small but then require complete consistency reevaluations. There are, however, a few noteworthy exceptions. The tool xLinkIt proposed by Nentwich et al. [10] provides an incremental way to finding inconsistencies among UML diagrams. They did this by parsing the consistency rules to infer how they are affected by model changes. xLinkIt is, thus, restricted to a simple language for writing consistency rules—though a language that is adequate in many situations. However, xLinkIt cannot handle the rather complex abstraction productions needed for maintaining consistency among multilevel class models. Wagner et al. [37] provide another approach to incremental consistency checking. They establish relationships between a model change and a consistency rule. Once a change is caught, the tool can recheck the associated rule to find inconsistencies caused by the change. However, the approach requires a correct and complete list of how changes affect consistency rules—this approach is, thus, only really usable for simple consistency rules. Our approach does require knowledge on how high-level and low-level classes are related but it does not require knowledge on how changes affect consistency rules. Furthermore, our approach maintains knowledge of previous transformations (abstracted low-level class models are not discarded but incrementally updated).

3 Background

In the first Section 3.1, we introduce the consistency requirements that we will apply throughout this paper and then apply a simple hotel example to illustrate how consistency checking is performed. In the Section 3.2, we briefly introduce the rules that will be used to support class abstraction.

3.1 Consistency for Class Model Refinement and an Example

In order to demonstrate the consistency problem for class model refinement, we use an illustrative example in the remainder of this paper. The example is based on a simplified hotel management system (HMS) taken from [12], where an engineer describes the system via two class models at different levels of abstraction. The purpose of the HMS is to provide support for hotel reservations, check-in/check-out procedures, and associated financial transactions.

Imagine that the engineer first concentrates on the class model at a high level of abstraction. In that context, a guest may have a reservation at a hotel or stay there, and the guest may have expense and payment transactions associated with such reservations and/or stays. This simple, high-level class structure is shown in Fig. 1 (top). Over time, the engineer may find it necessary to add more details to the class model. The result is the second class model, shown in Fig. 1 (bottom), which depicts a refinement of the first one. In the second class model, a guest was defined to be a person (inheritance) and every person was assigned an account—an account that supports transactions such as payments or expenses. Moreover, *Room* and *Reservation* classes were added to better define the difference between reservations and room occupancy.

Today, engineers often refine a high-level class model by adding lower level details—hence, overwriting the high-level model. Unfortunately, engineers then lose the higher level model—a model that is not only easier to understand (i.e., for later maintenance or training) but might also have been used to validate desirable properties (i.e., deadlock, performance, security) and the correct and complete implementation of requirements. It is, thus, beneficial to

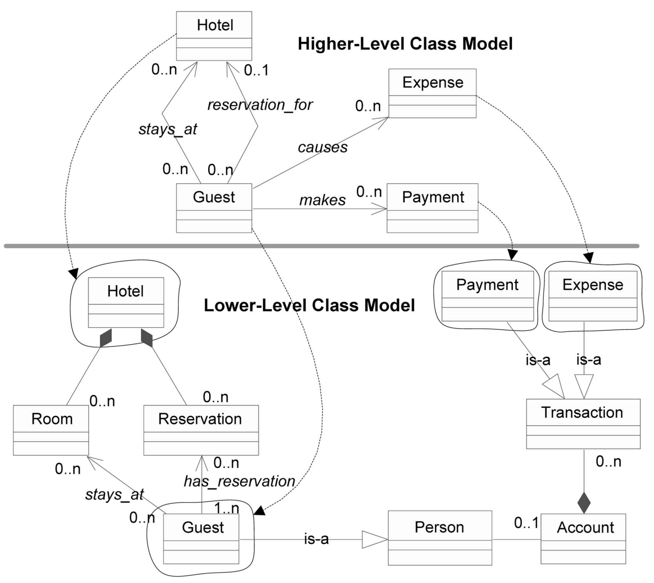


Fig. 1. Refinement of a class model.

maintain high-level class models (HCMs) and low-level class models (LCMs) separately. However, doing so is only then useful if these class models remain consistent over time. Two class models at different levels of abstraction are said to be consistent if the following consistency requirements are satisfied:

- Every low-level class (abbreviated as LCM class) refines at most one high-level class (abbreviated as HCM class): ensures that a low-level class refines one and only one high-level class.
- Every high-level class has at least one low-level class, which refines the high-level class: ensures that every high-level class is refined.
- The group of relationships between any two high-level classes must be identical with the group of relationships between their corresponding low-level classes: ensures the same interactions among low-level classes and high-level classes.

We must also introduce the representation of the refinement relationship in our approach. While name similarity of classes at the different levels of abstraction may imply a refinement relationship, engineers can also choose classes with different names. Thus, we use an explicit trace dependency to show the refinement relationship between a high-level class and a low-level class. In Fig. 1, trace dependencies are indicated as dashed lines.

The first requirement checks whether a low-level class refines at most one high-level class. An LCM class traced by a high-level class is called an *important class*; otherwise, an LCM class without any trace relationship is called a *helper class*. In Fig. 1, important classes are accentuated with circles—note that each important LCM class refines one high-level class only. Since every LCM class refines at most one HCM class, we say that the class models in Fig. 1 satisfy the first requirement.

The second requirement checks whether all HCM classes have an LCM class that refines them, respectively (completeness). This requirement also allows multiple LCM classes to refine an HCM class [13], however, this issue is not discussed further since the aggregation of multiple LCM classes into a single HCM class is not computationally expensive. Revisiting the example, we thus validate whether the four HCM classes *Hotel, Guest, Expense*, and *Payment* in Fig. 1 have a corresponding low-level class. It is easy for readers to see that each of these four HCM classes is refined by at least one low-level class, respectively, via the trace dependency.

The third requirement checks the similarity of the relationships among the classes. To do this, we need to see whether the relationships among the important LCM classes are equivalent to the relationships among their corresponding HCM classes. This is rather complex because relationships among important LCM classes are obscured through the presence of helper LCM classes. For example, the HCM defines that a *Guest* can make *Payments* whereas the LCM defines that a *Guest* is a *Person* who is allowed to make *Transactions* of which one kind of transaction is a *Payment*. So, are these two statements equivalent?

To ensure requirement 3, we, thus, need to find the derived relationships between any two important classes and these derived relationships should reflect the meaning of the path of helper classes between the two important classes. In order to achieve this, we apply abstraction productions, which will be discussed below. After the abstraction, we compare whether the two groups of relationships are identical. Two groups of relationships are identical if both groups have the same number of relationships and each relationship is structurally equivalent to one in the other group. In this paper, we focus on the type and direction of a relationship only. In [12], [13], we demonstrated that we can handle other properties as well (e.g., multiplicity).

The validation of the third requirement is not trivial and it is also the most expensive one to compute. For example, when we consider whether the class models in Fig. 1 satisfy requirement 3, we first need to derive all transitive relationships among the important classes *Hotel, Guest, Payment*, and *Expense*. After that, we need to compare whether the group of relationships between any two high-level classes is identical to the group of derived relationships between their corresponding low-level classes. For instance, we compare the group of relationships between high-level classes *Hotel* and *Guest* with the group of derived relationships between low-level classes *Hotel* and *Guest*.

Doing this abstraction and comparison on small class models is not hard. However, when class models include hundreds of classes and relationships, it is virtually impossible for humans to identify and track all inconsistencies [11]. Thus, a tool-based automatic support would be of great help to engineers.

There are a few additional consistency requirements that pertain to the specific properties of classes and relationships, but these are too distracting to consider here and are omitted. Obviously, our approach requires the existence of at least two levels of class models—although more than two may exist. The approach does not require any level to be guaranteed correct. Instead, the levels are compared pairwise to identify inconsistencies. It is the engineers' task to create and maintain class models and resolve inconsistencies. While we help the engineer identify inconsistencies quickly, we do not impose any constraints on when or how to resolve those inconsistencies [38], [39].

Our approach to consistency checking of class models is rule based in order to ensure correctness [12]. In a rule-based approach, every rule is instantiated by a collection of rule instances, whose existence is decided by the instances' respective *root element*. Additionally, every rule instance is associated with a *change impact scope*, which defines all model elements that potentially affect the validity of a rule instance—and, thus, the consistency between the two class models.

There are two kinds of rules in our approach. One kind of rules is called consistency rules, which directly support a consistency requirement. The other kind of rules is called action rules, which are responsible for the maintenance of the intermediate, abstracted class models. For example, in Fig. 1, an action rule creates an association between LCM classes *Guest* and *Hotel* because the path between these two classes via class *Reservation* is abstractable via an abstraction production (discussed below). Action rules thus support consistency rules by helping maintain the abstract-level class model (ACM) consistent with the LCM to subsequently simplify the comparison between the ACM and HCM. Two class models at different levels of abstraction are then consistent if all consistency rules return true. Central to our rule-based approach is how to maintain change impact scopes to minimize the computational overhead. Details about rules, rule instances, and their change impact scopes will be further illustrated in the following text.

3.2 Abstraction Technique

As stated in the previous section, validating the third requirement is the most (computationally) expensive part of our consistency checking approach. This part relies extensively on abstraction. Abstraction is the process of simplifying a given class model by hiding details. However, abstraction is not about omitting model elements. If details of no interest were omitted, then the abstracted class model would be incomplete. Abstraction productions are needed to reinterpret the hidden information such that the abstracted class model summarizes their effect. For example, we know from Fig. 1 that *Hotel*, *Guest*, *Payment*, and *Expense* are the important classes of the LCM because these are the ones singled out in the HCM (through trace dependencies). Classes such as *Room* and *Person* are helper classes that are not depicted in the HCM. However, the helper classes are implied there—for example, the class *Room* is implied in the high-level relationship *stays_at*.

Our abstraction technique reinterprets the helper classes and their relationships based on a set of abstraction productions [12]. We apply a previously developed abstraction technique built together with IBM Rational. The technique takes an arbitrary complex class structure and derives so-called transitive relationships among its classes. A transitive relationship is the result of combining a collection of relationships via abstraction productions. Consequently, a transitive relationship is the semantic equivalent of a collection of existing relationships. For

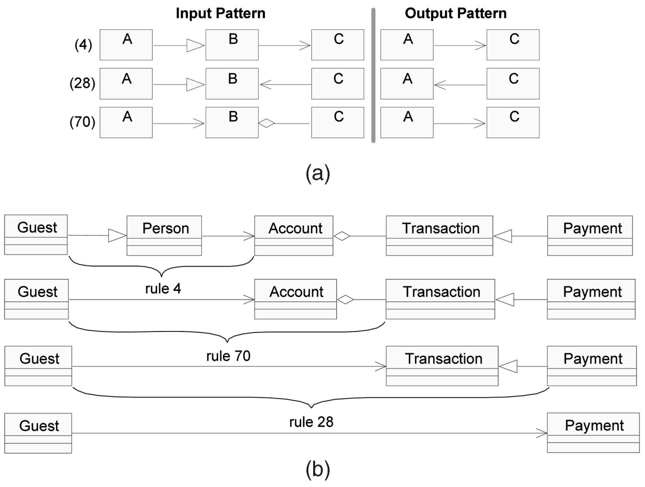


Fig. 2. Abstraction productions and an application of the productions. (a) Abstraction rules for UML class diagrams. (b) An application of the above rules.

example, if A has an association to B and B has an association to C, then, transitively, A has an association to C.

By composing the properties of a collection of direct relationships, one can infer properties of the transitive relationship. Properties of relationships include the direction and type of a relationship or the cardinality of association ends. Fig. 2a shows three abstraction productions out of the set of 121 productions defined in [12]. For instance, production 4 states that if A inherits from B and B calls C (input pattern), then, transitively, A calls C (output pattern). Or production 70 states that if A calls B and C is a part of B (diamond head), then, transitively, A calls C.

The abstraction rules were validated in [12] on 12 models with model sizes of up to several hundred classes. It was shown that the abstraction rules prevent false negatives (FN) but not false positives (FP). The lack of an abstraction thus truly means that no abstraction exists. However, not all abstracted paths do exist. That is, calling relationships among classes are typically conditional which makes understanding transitive relationships more complicated. Imagine that A calls B under a certain condition only, say, if variable x is true; and imagine that B calls C under the condition that x is false. In this case, A can never call C even though there is a transitive relationship from A to C. UML class models typically are not rich enough to understand conditions under which calls happen. As such, abstraction rules may err on the side of identifying calling relationships that are in fact wrong (false positives). In previous work, we found this error to be small with 4 percent of abstracted relationships only.

The abstraction productions are simple in nature. They describe a collection of two input relationships that are composable into a single output relationship (or not composable if the output pattern does not have a relationship). What makes the abstraction technique powerful is the large number of productions (121 productions for three types of class relationships and various properties). As input, the abstraction technique takes an arbitrary complex class structure and a list of important classes. The list of important classes emphasizes the classes that should not be

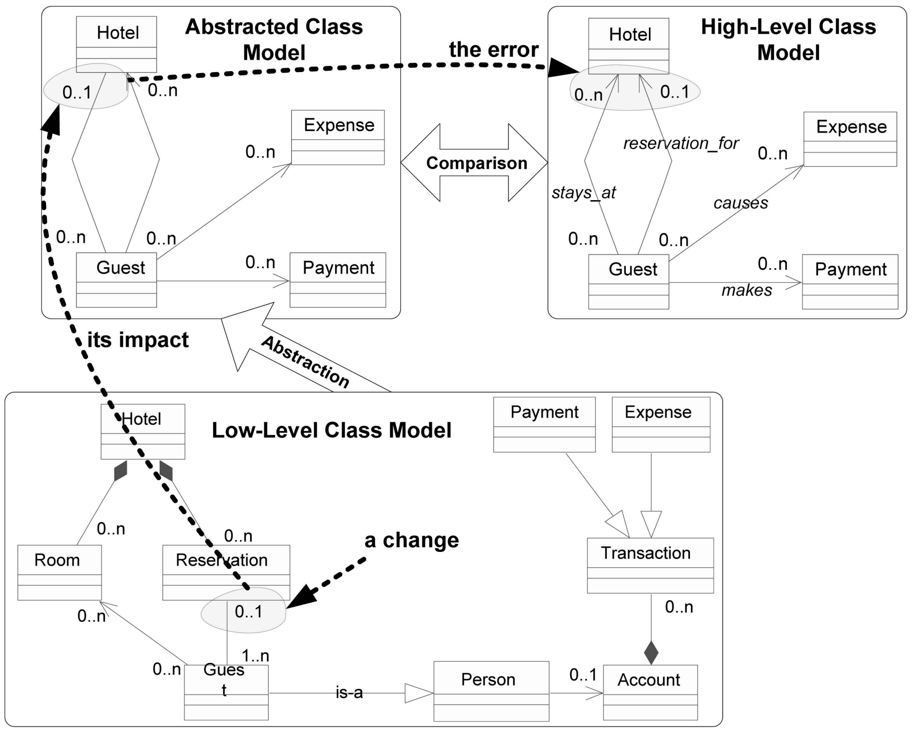


Fig. 3. The LCM is abstracted and the result is compared with the HCM after a change.

hidden. In Fig. 2a, classes *A* and *C* are important and the class *B* is not (i.e., the helper class) as it gets replaced together with its relationships by a transitive relationship.

Fig. 2b shows the use of transitive reasoning in understanding the relationship between the important classes *Guest* and *Payment* (from Fig. 1). Although the two important classes are not directly related to one another, a transitive relationship can be derived by eliminating the helper classes *Person*, *Account*, and *Transaction*. Fig. 2b shows that the application of production 4 eliminates the class *Person*, the subsequent application of production 70 eliminates the class *Account*, and, finally, the application of production 28 eliminates the call to *Transaction*. This technique uses abstraction productions incrementally to eliminate multiple helper classes. The final abstraction shows a single relationship between *Guest* and *Payment* that is semantically equivalent to the path of the helper classes. There are also productions such as those dealing with relationship properties like cardinalities. We ignore these productions in this paper for brevity. For more details, refer to [12]. We also ignore those productions that deal with the composition of multiple low-level classes into high-level classes because these productions are not computationally expensive [13].

4 Supporting Class Model Refinement

For simplicity, we concentrate on the correct refinement between two class models only—a single HCM and LCM. Dealing with the refinement of multilevel class structures is simply the concatenation of dealing with the refinement of any two adjacent levels. As was discussed above, the first two consistency requirements are easy to check while the third requirement requires the most computation during consistency checking. Thus, the validation of the third requirement will be the main issue in this paper. In principle, when an engineer changes a class model (either the high-level or the low-level one), the abstraction technique takes the LCM, computes an abstract class model (ACM) from the LCM, and compares the ACM with the HCM. Differences between the abstracted class model and the HCM indicate incorrectness/inconsistency. If this incorrectness did not exist beforehand, then it was introduced by the latest change.

Fig. 3 illustrates the impact of a change to a simple relationship in the LCM. There, the engineer changes the relationship between classes *Guest* and *Reservation* such that the calling direction is now bidirectional instead of unidirectional (see the arrowhead in the bottom model). To check whether this change causes an inconsistency, we first abstract the LCM and then compare the abstracted class model (ACM; top-left) with the HCM (top-right). It is then easy to see that the change causes an inconsistency because the derived relationship between LCM classes *Guest* and *Hotel* via *Reservation* is bidirectional while the corresponding relationships between the HCM classes *Guest* and *Hotel* are unidirectional.

Of course, the impact of a change differs with the type of change. We distinguish three major types of changes that include six different events as follows:

- The LCM changes (e.g., add/remove classes or relationships).
- The HCM changes (e.g., add/remove classes or relationships).
- The traceability changes (e.g., add/remove trace dependencies between HCM and LCM).

TABLE 1
Complexities of the Batch Abstraction

Model Name	Model Size	Time (ms)	Explored Paths
HMS	133	1141	205
Visualizer	213	1832	306
VOD3	229	3430	900
GEO	234	4430	496
World	271	4124	936
iTalks	476	4153	426
DSpace	780	14809	10101
OODT	1868	26969	7881

In principle, the abstraction-based approach to consistency checking could handle any of these events by reabstracting an LCM with the latest data. Doing so in its entirety is, however, not scalable as is discussed next.

4.1 Problem

Although the abstraction technique is reasonably scalable (see empirical studies in [12]), it is not instant, and thus, not able to keep up with an engineer's rate of model changes in real time. To compare the HCM with the LCM, a batch abstraction would have to explore and abstract all paths among all important classes. Table 1 shows the complexities of batch abstraction in terms of time and the number of paths explored. A model size is given as the sum of the LCM and HCM in the model. Here, we observed abstraction times of up to 27 seconds (with an average of 7.6 seconds, evaluated on a Dell computer with a 2.0 GHz CPU and 512 MB memory) for the eight sample models. These times are not unreasonable for occasional abstraction and consistency checking, but are not practical for instant use.

The obvious solution to our performance problem is to build an approach for incremental consistency checking. Instead of reabstracting the entire LCM anew with the latest change, incremental consistency checking only abstracts and compares a portion of the LCM. For the portion that we neither abstract nor compare, we must also demonstrate that the three requirements remain unchanged. The reevaluated portion should be as small as possible, but it must be big enough to adequately cover the impact of all possible changes.

4.2 Integrated Abstraction/Comparison (IAC)

To support incremental abstraction followed by necessary comparison, we initially implemented a tool that integrated incremental abstraction and incremental comparison (IAC). IAC performs a limited-scoped abstraction on demand after a design change and then compares the results with the HCM. IAC is implemented in an incremental fashion by instantiating rules based on relevant model elements. The following discusses how the three consistency requirements are implemented by three consistency rules in IAC.

Requirement 1: Every low-level class refines at most one high-level class.

$Rule1_{IAC}$ is implemented to directly support requirement 1. Since requirement 1 is interested in every low-level class, $Rule1_{IAC}$ is instantiated for every LCM class. Each rule instance evaluates separately whether its LCM class has a trace dependency to at most one HCM class. The root elements of instances of $Rule1_{IAC}$ are LCM classes because rule instances need to be created and destroyed as LCM classes are created and destroyed, respectively.

Each rule instance returns a Boolean value that reflects whether the root element satisfies the requirement or not. In order to support incremental consistency checking, each rule instance also has a scope, which contains all model elements that might change the validity of the rule instance. The reevaluation of a rule instance is necessary only if one or more of the model elements inside its scope change. In general, the change impact scope of a $Rule1_{IAC}$ instance contains its root element, an LCM class. It may also contain its trace dependency and the HCM class if the root element is an important class. Also refer to [11] for further details on rule instances and their scopes.

For example, in Fig. 1, we create an instance of $Rule1_{IAC}$ for the low-level class *Hotel*, denoted by $Rule1_{IAC} < Hotel >$. The rule instance $Rule1_{IAC} < Hotel >$ is created when class *Hotel* is created and the instance is destroyed when class *Hotel* is destroyed. Since there is a trace relationship between the high-level class *Hotel* and the low-level class *Hotel*, the change impact scope of the instance $Rule1_{IAC} < Hotel >$ includes three elements: the high-level class *Hotel*, the low-level class *Hotel*, and the trace dependency connecting these two classes. Once one of the elements is changed, such as the deletion of the trace, the instance $Rule1_{IAC} < Hotel >$ must be reevaluated to check whether the requirement is still satisfied. In [11], we demonstrated that this approach to incremental consistency checking is quite scalable for many kinds of consistency rules.

The evaluation of rule instances follows an automated, two-step process. The first step finds out whether new rule instances should be created and whether existing rule instances should be reevaluated (selection step). The second step evaluates all selected rule instances (evaluation step). We separate selection from evaluation because model changes made by engineers typically come in batches. For example, the deletion of the LCM class Y in Fig. 4 also causes the deletion of all its relationships (to X and Z). There are at least three changes caused by this single user action. By separating selection from evaluation, it is ensured that rule instances are reevaluated exactly once—thus,

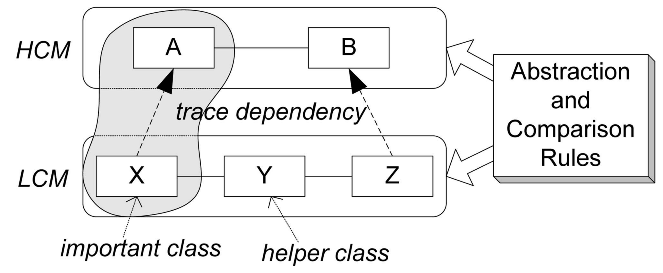


Fig. 4. A simple HCM, LCM, and traces.

saving computational effort. This separation also has the advantage that the rule instances could be evaluated in a different order than selected (this feature will become important later).

Requirement 2: Every high-level class has exactly one low-level class that refines the high-level class.

$Rule2_{IAC}$ realizes requirement 2 in IAC. This rule ensures that every HCM class has a trace dependency to exactly one LCM class. We chose an HCM class as the root element of an instance of $Rule2_{IAC}$. Thus, $Rule2_{IAC}$ instances are created and destroyed according to the existence of HCM classes. After a rule instance is created, it is responsible for evaluating whether requirement 2 is satisfied or not by the root element.

There are four HCM classes in Fig. 1. So, $Rule2_{IAC}$ is instantiated four times—once for every high-level class—and all four instances are validated initially to ensure consistency. For instance, $Rule2_{IAC}$ instantiated on HCM class *Hotel* (in short, $Rule2_{IAC} < Hotel >$) returns true because HCM class *Hotel* indeed has a trace to at least one LCM class.

The change impact scope of a $Rule2_{IAC}$ instance includes the HCM class, the corresponding LCM class, as well as their trace dependency if the HCM class was refined. Thus, the change impact scope of $Rule2_{IAC} < Hotel >$ is the same as the scope of $Rule1_{IAC} < Hotel >$ in Fig. 1.

Requirement 3: The group of relationships between any two high-level classes must be identical with the group of derived relationships between their corresponding important classes.

Starting from a high-level class, we can find its neighbor classes by traversing the direct relationships connecting to the high-level class. Let us first consider an HCM class X. For X, the investigation of a group of relationships between X and one of its neighbor HCM classes, say Y, consists of two steps. One is to find the two important classes corresponding to X and Y, and then derive all transitive relationships between the two important classes. The second step is to compare the two groups of relationships. Once the above two steps are completed, we can conclude whether the group of relationships between X and Y is identical with the group of relationships between their corresponding important classes. The algorithm that summarizes the above steps is as follows:

```
for each HCM class X do {
        for each X's neighbor class Y do {
        store all relationships between X and Y in set S
        find the corresponding LCM classes for X and Y,
          denoted by x and y
        abstract all paths between x and y and store all
          transitive relationships in set s
        compare the two sets S and s to see whether they are
          identical
    }
}
```

We choose an HCM class as the root element for $Rule3_{IAC}$ instances. Initially, our approach instantiates all $Rule3_{IAC}$ instances and validates whether their root elements satisfy requirement 3. When a change to a model element is caught, we only reevaluate those $Rule3_{IAC}$ instances, where the validity of requirement 3 on their root elements might be affected.

Now, it must be stressed that incremental abstraction is expected to abstract the transitive relationships of the LCM on demand. In other words, when $Rule3_{IAC} < X >$ is evaluated, the rule instance investigates the relationships between X and all its neighbor classes. The comparison, thus, triggers the abstraction of all transitive relationships between X's corresponding LCM class and all X's neighbors' corresponding important classes. Let us consider the rule instance for the high-level class *Hotel* in Fig. 3. $Rule3_{IAC} < Hotel >$ first finds all neighbor HCM classes through the HCM relationships. The HCM class *Hotel* only has one neighbor HCM class *Guest*. The rule instance then identifies *Hotel*'s corresponding low-level, important class by using the trace dependency. The important class is the name-equivalent *Hotel* in the LCM. Starting from the LCM class *Hotel*, the rule instance then searches and abstracts all possible paths of relationships until it reaches the LCM class *Guest*. Since there are two paths between the LCM classes *Guest* and *Hotel*, both paths are abstracted separately and the results are compared with the two HCM relationships between the HCM classes *Hotel* and *Guest*. The rule instance returns true only if the abstracted relationships are identical with their corresponding HCM relationships. However, $Rule3_{IAC} < Hotel >$ in Fig. 3 evaluates to false because the transitive relationship from the LCM classes Guest to *Hotel* via *Reservation* (the top-left of Fig. 3) is bidirectional while both relationships in the HCM (the top-right of Fig. 3) are unidirectional.

The change impact scope of $Rule3_{IAC} < Hotel >$ includes all model elements visited during the above evaluation. So it contains the HCM classes *Hotel*, *Guest*, and both of their relationships, the LCM classes *Hotel*, *Room*, *Reservation*, *Guest*, and all of their relationships; and the two trace dependencies. Fig. 5 depicts this impact scope for $Rule3_{IAC} < Hotel >$ graphically (the darker shaded area). A change to any element in this change impact scope results in the reevaluation of $Rule3_{IAC} < Hotel >$.

4.3 Observations of IAC

The implementation of $Rule3_{IAC}$ does not appear unreasonable on first glance, but readers must keep in mind that this is a small illustrative example only. Investigating paths from an important class to all its neighbor important classes is not cheap. Table 1 previously presented the cost of batch abstraction. There, the average number of investigated paths is 2,656. With an average of 136 explored

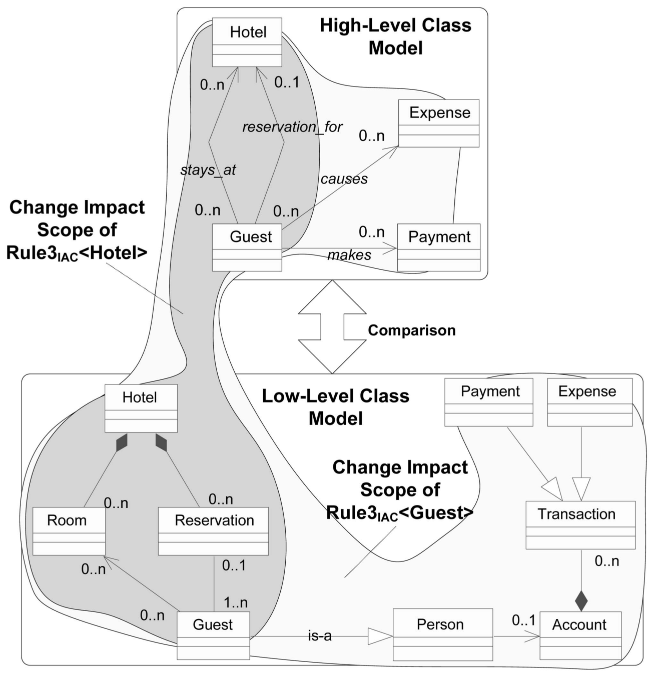


Fig. 5. The overlapping change impact scopes for $Rule3_{IAC} < Hotel >$ and $Rule3_{IAC} < Guest >$.

paths per user action, the IAC implementation is certainly an improvement over the batch abstraction presented earlier. However, we found two reasons why IAC is unnecessarily expensive:

1. The overlap of change impact scopes triggers duplicate reabstractions.

   There exist many rule instances whose change impact scopes have some common elements. While we conservatively add elements that might change the validity of rule instances, the common elements in different change impact scopes cause some paths to get abstracted more than once—an unnecessary burden. For example, in Fig. 5, the rule instance $Rule3_{IAC} < Hotel >$ investigates all relationships between high-level classes *Hotel* and *Guest* and abstracts all paths between their corresponding important classes. Similarly, the rule instance $Rule3_{IAC} < Guest >$ investigates relationships from *Guest* to all its neighbor classes. This includes the same paths that are considered by $Rule3_{IAC} < Hotel >$ plus the ones from *Guest* to *Payment* and *Expense* (the lighter shaded area in Fig. 5). Since the change impact scope of $Rule3_{IAC} < Hotel >$ is a subset of that of $Rule3_{IAC} < Guest >$, a change to the relationship between LCM classes *Guest* and *Reservation*, in the darker shaded area in Fig. 5, causes not only the reevaluation of $Rule3_{IAC} < Hotel >$ but also the reevaluation of $Rule3_{IAC} < Guest >$ because this relationship is in the scopes of both rule instances. The paths between the low-level classes *Hotel* and *Guest* are then abstracted twice.
2. Certain changes do not require reabstractions.

   We also discovered some unnecessary abstraction related to the nature and impact of changes. For example, if the HCM changes, say, one of the relationships between HCM classes *Guest* and *Hotel* in Fig. 5 is deleted, then the abstraction of the LCM is not affected because the LCM did not change. Yet, the IAC approach abstracts always. Likewise, if a new relationship, say, from *Guest* to *Hotel*, is added to the LCM, then more paths become available. But, none of the existing paths between *Guest* and *Hotel* are affected by this addition and so there is no need to reabstract them. Yet, the IAC approach does reabstract.

Perhaps one could tweak the implementation of $Rule3_{IAC}$ such that it understands these conditions. However, this solution is dangerous. We have discovered in [40] that augmenting incremental rules with "if" structures results in a combinatorial state explosion problem. There are simply too many changes and change combinations and it is very hard to validate the correctness of the consistency rules if they are tweaked—though it could be done. Also such an "if" structure is harmful to the Open-Closed Principle, which means that software entities should be open for extension, but closed for modification [41].

While tweaks and hacks might solve this problem, the fundamental issue is that we failed to recognize the different needs of comparison and abstraction (the IAC tool uses the same change impact scope to cover both). Since the consistency rules tightly integrate abstraction and comparison, they cannot be optimized independently of one another.

4.4 Separated Abstraction/Comparison (SAC)

To counter the computational inefficiency of the IAC, we investigated ways of separating abstraction from comparison. For separation, we use an intermediate model, called the abstract class model (ACM), such that abstraction maintains the correctness of the ACM relative to the LCM at all times and comparison ensures the correctness of the ACM with respect to the HCM. HCM and LCM are no longer compared directly and abstraction is no longer invoked by comparison as in IAC, as shown in Fig. 6. To compare this alternative approach to IAC, we developed another tool, called SAC.

The use of an intermediate model to simplify consistency checking is not new. Many existing approaches rely on intermediate models for consistency checking. However, thus far, intermediate models have been used solely 1) to unify among the different modeling languages involved ([23], [24]) and 2) to provide a formal foundation for the comparison rules ([43], [31], [33], [34]). In both cases, intermediate models have drawbacks because:

- They add another modeling language in addition to the ones already used.
- They cause discontinuity in that information has to be transformed to them (to detect the error) and back (to understand the error).
- They consume resources—memory and CPU.

However, the use of an intermediate model in SAC is new because it has not been recognized that intermediate models can serve as a foundation for separating incremental transformation from incremental comparison—with highly

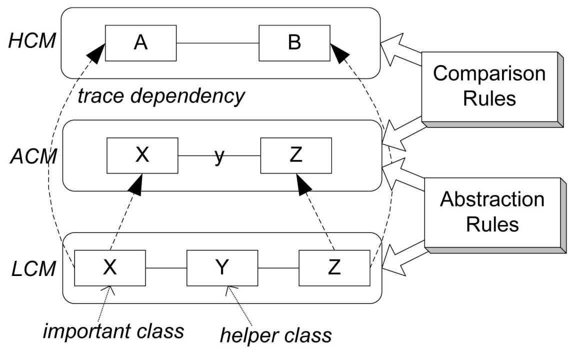


Fig. 6. Abstraction and comparison separated through an intermediate class model, the ACM.

beneficial results as are discussed below. Our use of an intermediate model is also different from traditional approaches because, here, the intermediate model is just another class model—we do not introduce a new modeling notation. Because of this reason, we do not cause (as much) discontinuity because the intermediate model is understandable by an engineer. However, it does consume memory. It should be noted that the use of an intermediate model does not make it easier for an engineer to understand inconsistencies when they happen. Both IAC and SAC can visualize the HCM and LCM elements involved in inconsistencies (by using traceability information and scope elements). SAC additionally can visualize the intermediate model. However, in both cases, it is not obvious how to best visualize this information to the engineer to support the understanding and resolving of inconsistencies. This issue is out of the scope of this paper.

Since the first two consistency requirements were computationally cheap, we reimplemented $Rule1_{SAC}$ and $Rule2_{SAC}$ similar to the corresponding rules in IAC. The only difference is that $Rule1_{SAC}$ also maintains ACM classes. We skip the discussion of requirements 1 and 2 to concentrate on the problematic requirement 3.

The significant difference between IAC and SAC is the introduction of an ACM. The comparison between two class models at different levels, required for consistency checking, is then divided into two kinds of rules. The abstraction rules are responsible for reabstracting the LCM if it changes and the comparison rules are responsible for comparing the ACM and HCM. In our case, abstraction rules are further subdivided in order to separate the search for abstractable paths from the maintenance of ACM relationships: If the LCM changes, then an abstraction rule, called $Abstraction_{SAC}$, searches and abstracts all affected paths connecting important classes in the LCM (starting at the location of the change). If the abstraction rule finds an abstractable path, then it adds a transitive relationship to an ACM. A second rule, called $AbstractablePath_{SAC}$, then maintains these abstracted ACM relationships. Finally, if the ACM or HCM changes, then a third rule, called $Comparison_{SAC}$, compares HCM elements with ACM elements for consistency. The first two rules are action rules (for transformation) while the third one is a consistency rule (for comparison). Note that an LCM change only then leads to a comparison if it actually changes the ACM, and an HCM change never leads to an abstraction. An overview of supporting requirement 3 is given as follows:

```
for each change caught by SAC{
  if the LCM changed then {
      search and abstract the paths among the affected elements
          // done by $AbstractionRule_{SAC}$
      update the ACM if necessary
          // done by $AbstractablePathRule_{SAC}$
  }
  if ACM or HCM changed then
    compare the ACM and HCM
          // done by $ComparisonRule_{SAC}$
}
```

The abstraction rule $AbstractionRule_{SAC}$ is responsible for abstracting paths in an LCM. Unlike all the previous rules, we apply the singleton pattern [42] to create only one instance of $AbstractionRule_{SAC}$. The instance is created when SAC starts. There are two tasks for this rule: 1) to search for abstractable paths and 2) to abstract them when found. If the search finds an abstractable, transitive path between two important classes, a new relationship between their corresponding ACM classes is added to the ACM. $AbstractionRule_{SAC}$ also creates an instance of $AbstractablePathRule_{SAC}$ to maintain the newly added relationship in the ACM (more detail will follow later). $AbstractionRule_{SAC}$ does not have a change impact scope. Instead, it incrementally searches and abstracts LCM classes and relationships starting at the location of the change (we will see below that this rule is optimized by understanding the change event, and thus, applying previous abstraction results in the ACM).

It is not always necessary for $AbstractionRule_{SAC}$ to explore paths in the LCM. The search for a new transitive relationship is dependent on the type of change event. We identified a total of six types of change events, three of which can cause the creation of a transitive relationship between two important classes.

The first type of event is adding a trace to a helper class so that it is upgraded to an important class. In this case, the search algorithm of $AbstractionRule_{SAC}$ explores all paths from the upgraded class to its neighbor important classes to find whether new transitive relationships can be abstracted. This search algorithm is exactly the same as the one used by $Rule3_{IAC}$. The pseudocode to deal with adding a trace to a helper class is as follows:

```
if a change is adding a trace to a helper class x then
    add x' to ACM
    for each class y which is a neighbor class of x{
      search and abstract all paths leading to another
        important class via class y in LCM
      update relationships between y' and other ACM classes
        if abstractable paths found
    }
```

For example, in Fig. 1, suppose that the LCM class *Guest* was initially a helper class. If a trace is added to *Guest*, then starting from the new important class *Guest*, the search algorithm searches and abstracts all possible paths of relationships until it reaches other important classes such as *Hotel*, *Payment*, and *Expense*. The addition of a trace may, thus, add an ACM class and relationships. It can also delete existing ACM relationships, which are discussed later under $AbstractablePath_{SAC}$.

The second type of event is deleting a trace. As a result, an important class is downgraded to a helper class. In this case, the search algorithm in $AbstractionRule_{SAC}$ explores all paths to find all transitive relationships between any two important classes via the downgraded class. Actually, the algorithm optimizes abstraction by utilizing the previous abstraction results. Specifically, the algorithm only abstracts those ACM relationships that end in the downgraded (i.e., now deleted) ACM class (done by $Rule1_{SAC}$). The removal of a trace, thus, eliminates an ACM class (and its relationships) and may add new ACM relationships in its place.

```
if a change is deleting a trace between the LCM class y and an
  HCM class then {
  find the corresponding ACM class y';
  for any two neighbors x and z of y' {
        abstract the path between x and z via y';
        update relationships the ACM if necessary;
  }
  remove y' from ACM;
}
```

For instance, for the LCM shown in the bottom of Fig. 3, SAC produced an abstraction result, shown by the abstracted class model in Fig. 3 (top-left), based on the trace dependencies given in Fig. 1. If the trace connecting class *Guest* is deleted, then the search algorithm in $AbstractionRule_{SAC}$ only abstracts any two relationships affected by the deletion of *Guest* in the ACM. Specifically, the search algorithm tries to abstract the relationship between *Expense* and *Guest* and the relationship between *Payment* and *Guest* in the ACM (the top-left of Fig. 3). If they are abstractable, then a new transitive relationship is directly added between *Expense* and *Payment*. It is not necessary for the search algorithm to explore any affected path in the LCM. As such, SAC saves abstraction effort by taking advantage of the previous abstraction results stored in the ACM.

The third type of event is adding a new relationship to an LCM (between two existing classes). $AbstractionRule_{SAC}$ checks whether this new relationship connects two important classes. If neither class is important, then the search algorithm explores all transitive relationships between the affected classes and all neighboring important classes (avoiding any circularity through common classes or relationships). All transitive relationships from one important class are then combined with the transitive relationships from the other important class. During this combination step, $AbstractionRule_{SAC}$ chooses one transitive relationship from each side and combines/abstracts the two transitive relationships with the newly added relationship. If these three relationships are abstractable, then $AbstractionRule_{SAC}$ creates an instance of $AbstractablePathRule_{SAC}$, which, in turn, creates and maintains a new ACM relationship.

If the newly added relationship connects an important class, then $AbstractionRule_{SAC}$ skips the abstraction of paths starting from that important class. If the newly added relationship connects two important classes, then no abstraction is necessary and the newly added relationship is also added to the ACM. The following pseudocode

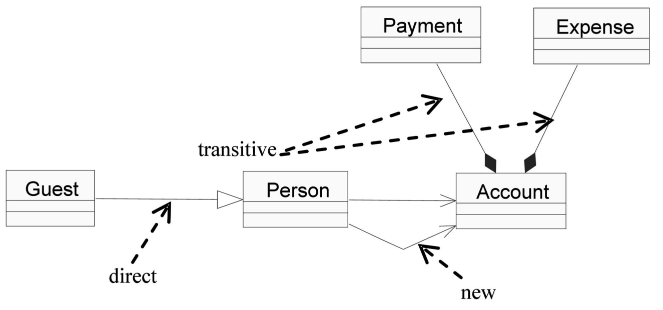


Fig. 7. After a new association from *Person* to *Account* is added.

summarizes how the algorithm deals with adding a new relationship in an LCM.

```
if a change is adding a relationship r between x and y in an
  LCM then {
      search and abstract all paths starting from x and leading
        to an important class and save them in set X;
                  // skipped if x is an important class
      search and abstract all paths starting from y and leading
        to an important class and save them in set Y;
                  // skipped if y is an important class
      for any abstractable path m in X
        for any abstractable path n in Y
        abstract paths m, r and n and update relationships in
          ACM if necessary
}
```

For example, let us consider the LCM in Fig. 1. Assume that a new association is added between classes *Account* and *Person* (Fig. 7). The search algorithm in $AbstractionRule_{SAC}$ initially finds abstractable paths via *Person* and *Account* because both are helper classes. The search algorithm then finds two more paths from *Account*. One path is connected to *Payment* while the other path is connected to *Expense*. Since both *Payment* and *Expense* are important classes, the search stops and the abstracted relationships are stored in the transitive relationship group for *Account*. The search algorithm then continues on the other end of the new association—the class *Person*—to explore all paths leading to important classes there. Since only one relationship exists and it ends in the important class *Guest*, there is only one relationship in the transitive relationship group for *Person*. The search algorithm then abstracts the transitive relationships identified by combining the paths from *Person* with the paths from *Account*. This results in two paths. One path includes class *Guest*, the generalization, class *Person*, the newly added association, class *Account*, the composition, and class *Payment* while the other path connects classes *Guest*, the generalization, class *Person*, the newly added association, class *Account*, the composition, and class *Expense*. After applying the abstraction productions, the search algorithm produces two new transitive relationships, and thus, two new $AbstractablePathRule_{SAC}$ instances.

$AbstractionRule_{SAC}$ creates new instances of $AbstractablePathRule_{SAC}$ for every new abstractable, transitive relationship between any two important classes. Thus, an instance of $AbstractablePathRule_{SAC}$ represents an ACM relationship, which is also the root element for

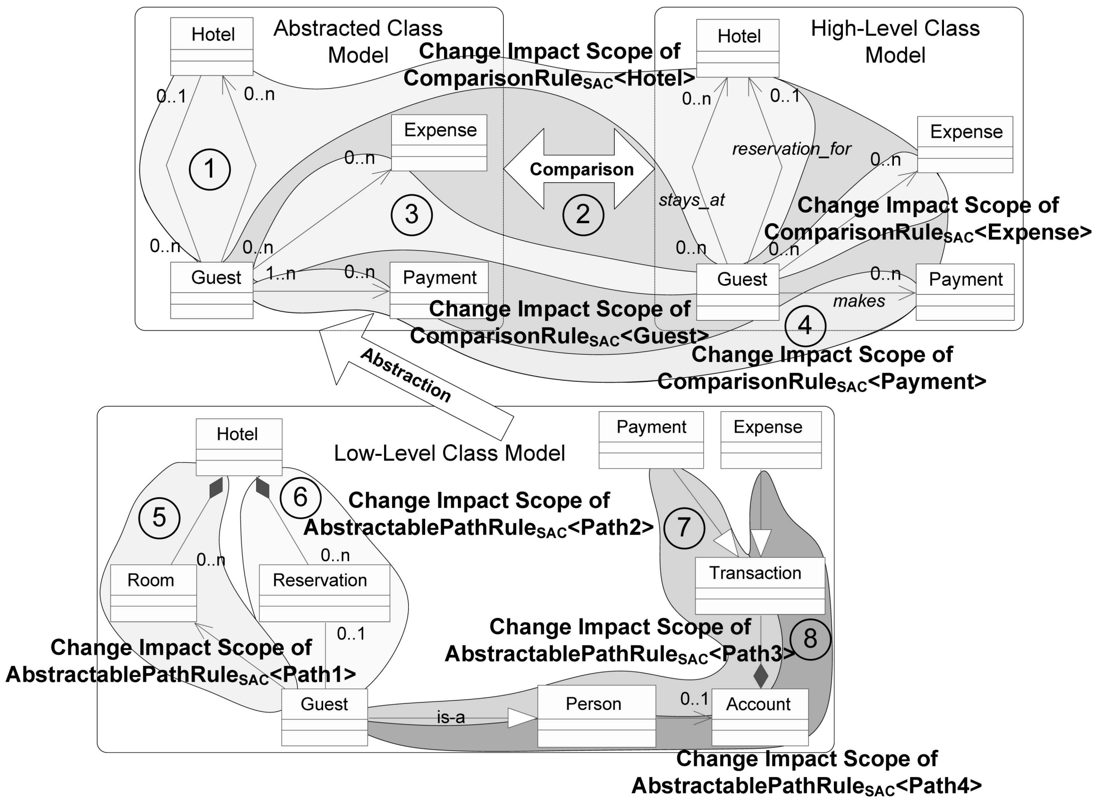

Abstracted Class Model
High-Level Class Model
Low-Level Class Model
Hotel
Expense
Payment
Guest
Room
Reservation
Person
Account
Transaction
Comparison
Abstraction
reservation_for
stays_at
makes
is-a
Change Impact Scope of ComparisonRuleSAC<Hotel>
Change Impact Scope of ComparisonRuleSAC<Expense>
Change Impact Scope of ComparisonRuleSAC<Guest>
Change Impact Scope of ComparisonRuleSAC<Payment>
Change Impact Scope of AbstractablePathRuleSAC<Path1>
Change Impact Scope of AbstractablePathRuleSAC<Path2>
Change Impact Scope of AbstractablePathRuleSAC<Path3>
Change Impact Scope of AbstractablePathRuleSAC<Path4>
1
2
3
4
5
6
7
8

Fig. 8. The change impact scopes of $AbstractablePathRule_{SAC}$ and $ComparisonRule_{SAC}$ instances.

the instance. For example, based on the new unidirectional association from class *Person* to *Account* shown in Fig. 7, a new unidirectional association from class *Guest* to *Expense* is created in an ACM. The change impact scope of an instance of $AbstractablePathRule_{SAC}$ includes all classes and relationships that are visited during the abstraction plus the two traces connecting the two important classes. A change to any element in the scope, such as the deletion of a relationship, causes the reevaluation of the instance of $AbstractablePathRule_{SAC}$. With a deletion, the reevaluation also deletes the corresponding relationship in the ACM.

$ComparisonRule_{SAC}$ compares the group of relationships between any two HCM classes with the group of relationships between their ACM counterparts. Just like in IAC ($Rule3_{IAC}$), SAC creates an instance of $ComparisonRule_{SAC}$ for every high-level class (i.e., the root element). The change impact scope of a $ComparisonRule_{SAC}$ instance contains its HCM class, all the relationships connecting the HCM class, and all the neighbor classes of the HCM class; the scope also contains the ACM class (corresponding to the root HCM class), all the relationships connecting the ACM class, and the ACM class' neighbor classes; finally, the scope contains all trace dependencies of the above-mentioned HCM classes (see the upper part of Fig. 8, which depicts the change impact scopes for the four rule instances of $ComparisonRule_{SAC}$ in four different numbers). Its scope does not include any LCM elements (i.e., an LCM change triggers a comparison only if the ACM changes).

Note that SAC still suffers from the duplicated comparison problem discussed in IAC. For example, $ComparisonRule_{SAC} < Hotel >$ compares the same two relationships of $Hotel$ as does $ComparisonRule_{SAC} < Guest >$. But, due to the separation of transformation and comparison, SAC no longer duplicates the expensive abstraction production (as will be demonstrated in Section 5.1). However, there is a restriction on the ordering of the above three rules related to the validation of the third requirement. Namely, $ComparisonRule_{SAC}$ should not be performed until all rule instances of $AbstractionRule_{SAC}$ and $AbstractablePathRule_{SAC}$ have completed their reevaluation.

4.5 Comparison between IAC and SAC

Both SAC and IAC respond to different types of changes in different ways. There are in total 13 types of changes and nine thereof require abstraction in case of IAC, whereas only five require abstraction in case of SAC. Table 2 summarizes the types of changes. Table 2 has some entries marked "Possible." These entries imply that certain types of changes may or may not require the evaluation of a $ComparisonRule_{SAC}$ rule instance because a change to the LCM may not always cause a change to the ACM. The SAC approach also compares less than IAC but these additional savings are largely insignificant for computational scalability because comparison is cheap.

Both IAC and SAC react to model changes and incrementally check the consistency between two class models (incremental abstraction followed by incremental comparison), but they do so in different ways, which result in different performances. When a rule instance is reevaluated, IAC rediscovers the low-level class model because, typically, consistency checking rules do not maintain state. SAC, on the other hand, separates abstraction from comparison by

TABLE 2
Changes Have Less Impact in SAC than in IAC

Type of Changes	Abstraction		Comparison	
	IAC	SAC	IAC	SAC
Add a HCM relationship	Yes	No	Yes	No
Delete a HCM relationship	Yes	No	Yes	Yes
Move a HCM relationship	Yes	No	Yes	Yes
Add a HCM class	No	No	Yes	Yes
Delete an isolated HCM class	No	No	No	No
Add a LCM class	No	No	No	No
Delete an isolated LCM class	No	No	No	No
Add a trace	Yes	Yes	Yes	Yes
Delete a trace	Yes	Yes	Yes	Yes
Add a LCM relationship	Yes	Yes	Yes	Possible
Delete a LCM relationship	Yes	No	Yes	Possible
Move a LCM relationship	Yes	Yes	Yes	Possible
Move a trace	Yes	Yes	Yes	Yes

maintaining an intermediate model—the abstracted class model (ACM). This intermediate model acts as a barrier, where abstraction rules create the intermediate model but do not care about its use and comparison rules use the intermediate model for consistency checking—though do not care about how it was created. Each set of rules is optimized for its own purpose by considering their own needs only. This is why IAC failed to be efficient because, by integrating abstraction and comparison, these rules could only be optimized to the *weakest of both needs*.

Note that SAC needs to distinguish the types of model changes in order to make the most use of an ACM. As such, only certain types of model changes can trigger the path search (i.e., only when new paths are created). In contrast, it does not make sense for IAC to distinguish these types of model changes because IAC does not save previous abstraction results and it has to reexplore a portion of the LCM in order to compare the abstraction results with the HCM. As for the implementation of the two approaches, the main difference between SAC and IAC lies in the implementation of requirement 3. Instead of one rule in IAC, SAC uses three rules—two rules dealing with changes in the LCM and the corresponding updating of the ACM, respectively; and one rule for the comparison between the HCM and the ACM.

In summary, it is important to separate incremental abstraction from incremental comparison so that transformation and comparison can react optimally to changes. This results in smaller change impact scopes. The smaller the scope, the less likely is the need to reevaluate these rules with model changes. SAC has much smaller scopes than IAC, and thus, performs much better. However, how better does SAC perform in real applications? We will answer this question in the next section.

5 Validation

In this section, we will compare SAC and IAC in terms of computational effort, memory consumption, and accuracy. Remember that IAC represents the traditional approach to consistency checking, where transformation and comparison are integrated. We, therefore, evaluated it together with SAC to better demonstrate the benefits of separating abstraction from comparison. We evaluated both IAC and SAC on eight third-party models. We evaluated both approaches by randomly injecting model changes (see Table 5) on the models—thus, varying the kinds of model changes (deletions, additions, and modifications) or the ratio of important classes to the helper classes (i.e., the most significant factor affecting the abstractions). In total, over 1,500 model changes were made and 14,000 rule instances were evaluated on a Dell computer with a 2.0 GHz CPU and 512 MB memory.

The empirical evidence shows that SAC not only is much faster and more accurate than IAC but also does not exhibit the scalability problems we saw with IAC, even on large models. At the same time, SAC does not consume more memory than IAC.

5.1 Computational Complexity

The following focuses on the computational cost of requirement 3 (requirements 1 and 2 are similar for IAC and SAC and computationally cheap). The implementation of requirement 3 consists of two steps. The first step abstracts the necessary paths between two important classes in a low-level class model. The second step compares the abstraction with the high-level model. For both approaches, abstraction is performed by starting from a class, exploring every path until it either reaches an important class or returns due to a nonabstractable subpath. Table 4 lists the average consistency checking time versus comparison time (in milliseconds) based on the eight models with different percentages of important classes (20, 40, 60, and 80 percent randomly chosen). The empirical data show that both IAC and SAC spend most of their time on path abstraction; the comparison time accounts for a very small portion of the total time (5 percent, on average). Therefore, the abstraction portion dominates the computational cost of consistency checking.

In the following, we thus focus on the computational complexity of abstraction and introduce some factors that help us understand it:

TABLE 3
Visited Paths versus Abstractable Paths

Model	IAC			SAC		
	Total Paths	Abstractable Paths	Percent	Total Paths	Abstractable Paths	Percent
HMS	55.39	13.46	0.24	7.60	1.52	0.20
Visualizer	210.78	21.17	0.10	14.86	2.47	0.17
VOD3	168.06	24.08	0.14	15.32	2.80	0.18
GEO	137.70	61.02	0.44	12.07	4.89	0.41
World	62.48	16.64	0.27	5.48	1.88	0.34
iTalks	69.38	25.66	0.37	8.28	2.99	0.36
DSpace	179.76	7.22	0.04	24.14	0.46	0.02
OODT	203.46	10.44	0.05	22.89	2.30	0.10

TABLE 4
Consistency Checking Time and Comparison Time in Milliseconds

Model	IAC		SAC	
	Total	Comparison	Total	Comparison
HMS	721.39	36.81	110.09	18.61
Visualizer	1124.47	28.01	98.19	10.94
VOD3	1075.73	37.14	117.81	7.02
GEO	1131.34	30.25	152.78	10.25
World	1030.91	17.93	92.72	10.85
iTalks	1128.13	19.98	102.43	8.56
DSpace	900.40	3.12	112.27	2.43
OODT	993.51	5.80	131.84	7.14

TABLE 5
Number of Affected Classes Per Model Change in IAC

	HMS	Visualizer	VOD3	GEO	World	iTalks	DSpace	OODT
#Classes	5.3	5.9	7.9	7.7	2.7	3.2	1.7	2.5

- *ClPE* represents the average number of classes from which we should abstract paths in response to one event.
- *RePCl* is the average number of relationships connected to a class.
- *PaLen* denotes the average path length from a class to an important class.

$RePCl^{PaLen}$ is the number of paths leading to other important classes from a class. In practice, once a portion of a path cannot be abstracted, the abstraction stops exploring the rest of the path. Since consistency checking is dominated by the computational cost of abstraction, we can, thus, say that the cost of consistency checking has the complexity $O(ClPE * RePCl^{PaLen})$ as the worst case.

In practice, the cost of consistency checking is much less than $ClPE * RePCl^{PaLen}$. Table 3 empirically shows the actual number of paths which can be abstracted versus the number of paths which are explored. While there is some fluctuation among the eight models, we found the average ratio of abstractable paths to all paths to be around 0.2 with the maximum ratio being 0.44. Intuitively, one would think that *RePCl* becomes larger when the size of the model increases. However, this did not turn out to be true. In fact, when the size of a model increases, the number of relationships such as associations connecting to a class should not become larger; otherwise, the structure/implementation of a class in a software system can become overloaded, which violates the low coupling principle employed during software development [43]. Therefore, in a well-designed system, such overloaded classes are rarely to be found.

To demonstrate the above idea, we empirically investigated the value of *RePCl* and found it to be bounded. In Fig. 9, all classes of the eight models are numbered at the x-axis and the y-axis represents the number of relationships for each class. We observed that 80 percent of the classes have between 1 and 10 relationships with the average number of relationships being 2.86 (red line). Therefore, we conclude that the value of *RePCl* is a moderate number and the value of *RePCl* does not increase with the size of a class model—an important scalability factor.

PaLen is another factor, which impacts the time performance of IAC and SAC. During class model refinement, the helper classes are typically added to the LCM to refine the relationship between two HCM classes. So, the path between a helper class and an important class is always shorter than the path between two important classes. In other words, *PaLen* is bounded by *LenPPa*, which is the average length of a

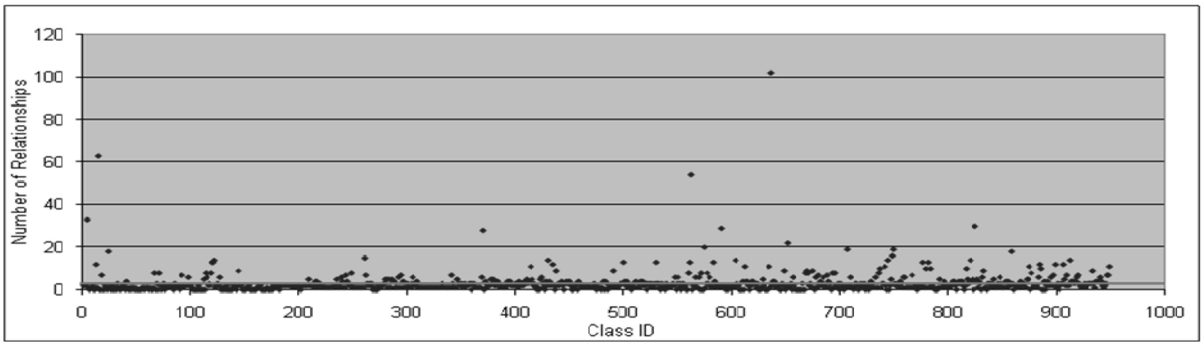


Fig. 9. *RePCl* is bounded.

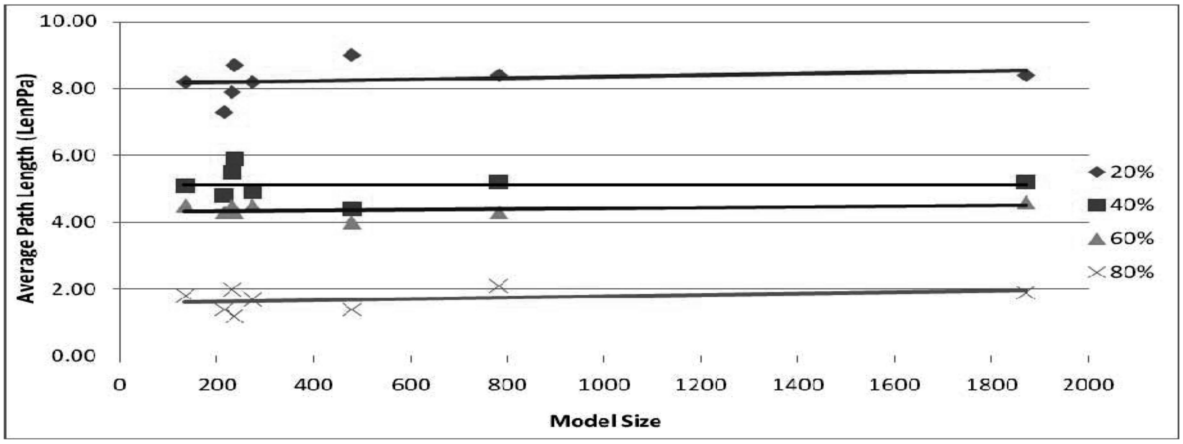


Fig. 10. *LenPPa* decreases with the percentage of important classes.

path between two important classes. Therefore, the computational complexity of consistency checking is bounded by $O(ClPE * RePCl^{LenPPa})$. One would again expect that the value of *LenPPa* increases with the size of a model. However, this expectation is once more contrary to the reality. We investigated *LenPPa* on the eight models and found that it depends solely on the percentage of important classes among LCM classes—the higher the percentage of important classes, the smaller the value of *LenPPa*.

To empirically demonstrate the relationship between *LenPPa* and the percentage of important classes, we investigated the values of *LenPPa* based on 20, 40, 60, and 80 percent important classes in the eight models. Fig. 10 shows that the value of *LenPPa* is not affected by the size of a model. Instead, the value of *LenPPa* depends only on the percentage of important classes (note that the value of *LenPPa* stays constant for a given percentage).

For instance, when there are only 20 percent important classes in the eight models, the value of *LenPPa* is, on average, 8.3 but when the percentage of important classes increases to 80 percent, the value of *LenPPa* is only 1.7. The value of *LenPPa* decreases with the percentage of important classes.

Above, we have shown that the value of *LenPPa* is dependent on the structure of a class model instead of its size and *RePCl* is bounded within a small range. Moreover, these two factors stay the same for both IAC and SAC for a given model. However, the value of *ClPE* shows how IAC and SAC work differently. Since each abstraction computation starts from an LCM class to find all abstractable paths leading to all its neighbor important classes, the number of LCM classes affected by a change decides how long the abstraction computation will take. In order to get the shortest abstraction time, we should have the smallest number of classes affected by an event. IAC has more rule instances affected by any change event because the large change impact scopes of $Rule3_{IAC}$ instances have a high probability of overlap. Consequently, the number of affected LCM classes in IAC is normally greater than that of SAC. Table 5 shows that, in the IAC approach, the average number of classes affected by a change event is between 1.7 and 7.9. Next, we illustrate that the corresponding value of *ClPE* in SAC is not only smaller but also the smallest for all types of events. Thus, we conclude that SAC is an optimal solution considering that the values of *RePCl* and *LenPPa* are not affected by the model size.

Recall that when events such as deleting a trace or relationship happen, no abstraction needs to be performed in a low-level class model. Consequently, the value of *ClPE* is 0, which is optimal. Only in the cases of adding a trace or a relationship, the value of *ClPE* could be nonzero. We discuss the two cases separately below. In the case of adding a trace, if the trace is added to an already important class *Y* (Fig. 11a), then it is not necessary to find any transitive relationships because the LCM class *Y* refines two HCM classes that violates the requirement $Rule1_{SAC}$. As a result, the value of *ClPE* is 0. If a new trace is added to the helper class *X* as shown in Fig. 11b, we should abstract all paths from this new important class to all other important

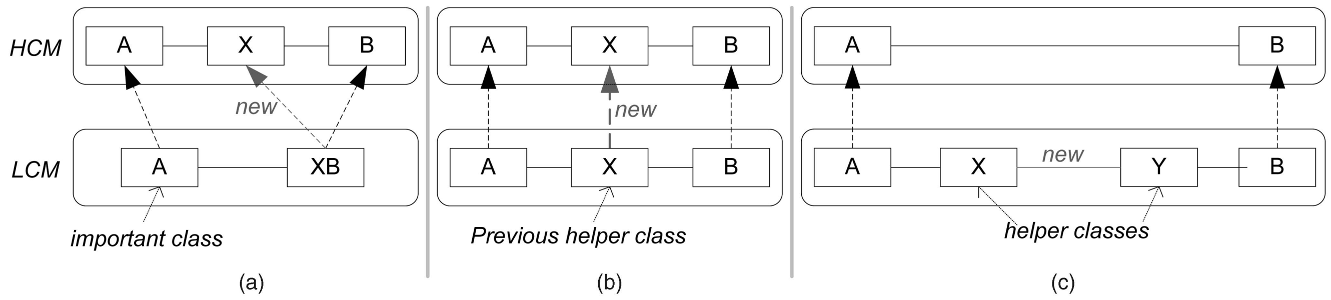


Fig. 11. How paths are explored in SAC for three different cases: (a) new trace is added to a previously important class; (b) new trace is added to a previous helper class; and (c) new relationship is added between helper classes.

classes (i.e., classes A and B in this case). Thus, the value of *ClPE* is 1.

When adding a relationship, there are several scenarios that affect the value of *ClPE*. If two classes connected by a new relationship are important, then the search for a new transitive relationship is not necessary since no new paths passing the newly added relationship can be created. The new relationship is simply added to the ACM and the value of *ClPE* is 0. If both classes connected by a new relationship are helper classes, then there could exist transitive relationships through the helper classes. For example, Fig. 11c shows such a new relationship added between helper classes X and Y. In this situation, some transitive relationships between important classes A and B might be created via X and Y. So, the abstraction algorithm should start from X and Y, respectively, to find new paths. Thus, the value of *ClPE* is 2, which is optimal. As a special case of the above scenario, if only one of the two classes is a helper class, the abstraction algorithm searches for new paths only starting from the single helper class. Compared to Table 5, it is obvious that the value of *ClPE* for SAC is much smaller than the corresponding value for IAC. The maximum number of affected classes in SAC is 2, which is the smallest possible number considering all types of events. Fig. 12 shows the time performance for both IAC and SAC in milliseconds. We see that SAC outperforms IAC by about an order of magnitude.

In summary, the value of *RePCl* is typically bounded within a small range and the value of *LenPPa* is only related to the percentage of important classes in a low-level class model. Consequently, the values of *RePCl* and *LenPPa* stay the same for both IAC and SAC for a given model and there is no scalability problem in terms of the model size. The only important scalability factor is *ClPE*. The smaller the value for *ClPE*, the more efficient the abstraction.

5.2 Memory Cost

While SAC improves time performance, it actually consumes less memory space than IAC and the memory consumption of IAC rises even more sharply than that of SAC. This observation was contrary to our intuition at first since SAC has to maintain an intermediate model with extra memory consumption. The memory cost of IAC ($Memory_{IAC}$) consists of two parts: the memory used to store all rule instances and all scope elements contained in all the rule instances' change impact scopes. The memory cost of SAC ($Memory_{SAC}$) consists of the same two parts as IAC, plus the memory space allocated for the intermediate model ACM. Since each rule instance is represented by its type such as $\mathrm{Rule2}_{<\mathrm{IAC}>}$ and a scope element is an element such as a class or a relationship in a class diagram, each rule instance and scope consume roughly the same unit in memory. So, in the following, we only consider the number of rule instances and the number all scope elements contained in them. While the intermediate model adds to the memory consumption of SAC, this cost is compensated by the significantly smaller scope sizes among SAC rules as compared to IAC. Fig. 13 illustrates the memory gain on the eight sample models with 20, 40, 60, and 80 percent

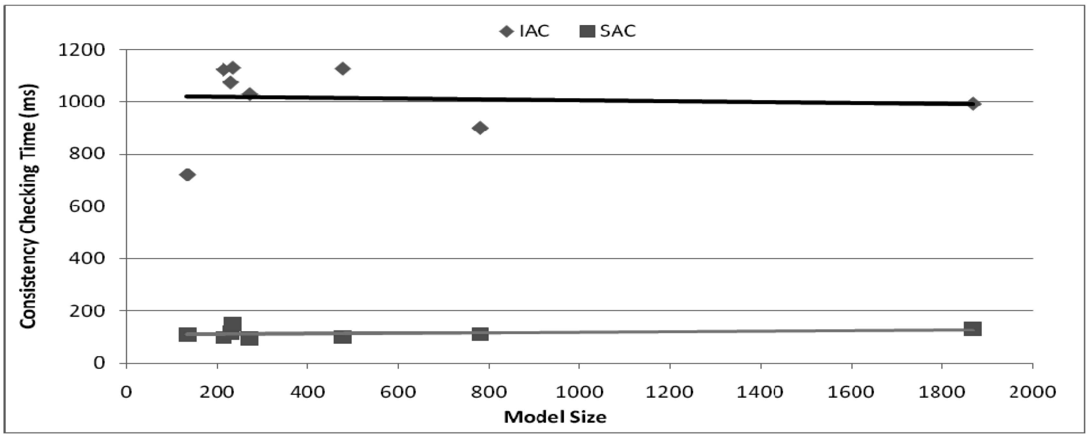


Fig. 12. Time performance of IAC and SAC.

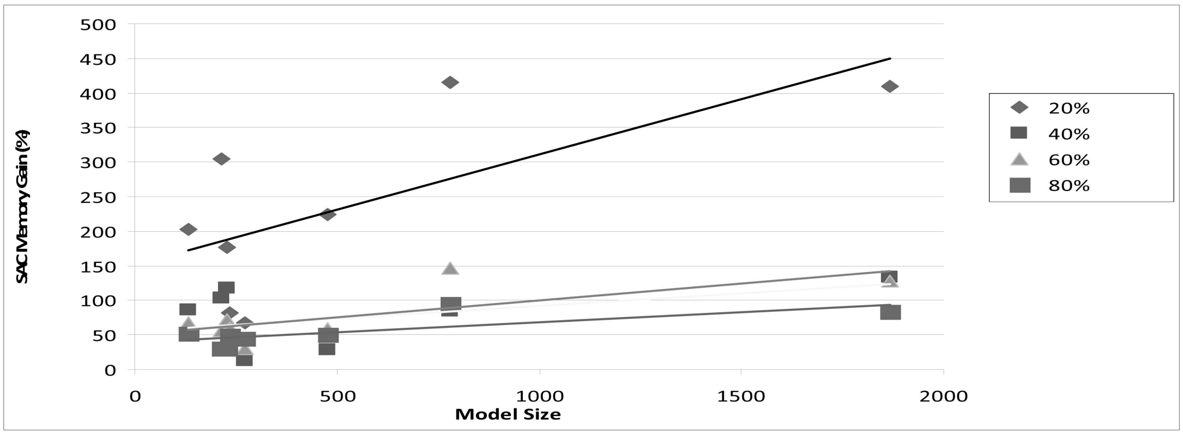


Fig. 13. Differences of memory costs: SAC uses less memory than IAC.

important classes. The memory gain is given by (IAC's memory—SAC's memory)/SAC's memory * 100, which represents the memory difference between IAC and SAC. In this figure, the x-axis shows each model in four percentages of important classes and the y-axis indicates the percentage of the memory gain between IAC and SAC for each percentage of important classes. We see that the percentages of memory gain for the eight models increase proportionally with the model sizes but decrease with the percentages of important classes.

5.3 Accuracy

We previously discussed that the abstraction technique errs in favor of abstracting too much instead of too little (i.e., false positives). We previously empirically evaluated the correctness of the abstraction results [12] and found the false positive rate to be less than 4 percent (100 percent sensitivity and 99.3 percent specificity). We also demonstrated that it was an extremely time-consuming task to infer abstract relationships among all class combinations and potential paths. That is, of the 21,024 potential dependencies among all 170 abstract classes of all 18 experiments in [12], there were only 2,374 paths and 258 transitive relationships—100-fold reduction. Both IAC and SAC implemented the same abstraction technique and it was expected that both approaches would do equally well in terms of the usefulness of the abstraction results. This was partially wrong. Not only was SAC computationally faster but it also produced fewer false positives as is explained next.

The existence of the 4 percent false positive rate implies the possibility of erroneous design feedback during refinement. In turn, 4 percent of the inconsistencies reported should be in fact wrong. We, thus, investigated how many abstract relationships were derived by both SAC and IAC. We discussed above that IAC unnecessarily reinvestigated many more paths than SAC. This should not have come as a surprise given that SAC reused abstraction results instead of reabstracting everything. We found that IAC explored, on average, 135.9 paths per change while the SAC explored only 13.8 (based on the same eight case studies and different percentages of important classes). Among the 135.9 paths explored by the IAC, 22.5 of them resulted in abstract relationships (i.e., not every path resulted in an abstract relationship). The SAC, on the other hand, only produced 2.4 abstract relationships, on average. Since we know of the existence of the 4 percent false positive rate, we can infer that IAC produced 9.4 times more false positives than SAC. Thus, while both IAC and SAC suffer from the 4 percent false positive rate, less abstraction (as in SAC) implies that fewer inconsistencies are detected through false abstraction results.

5.4 Tool Complexity

We implemented both IAC and SAC and integrated them with the design tool IBM Rational Rose. SAC implements more rules (transformation and comparison) than IAC (comparison only). In terms of implementation effort, it may thus appear that SAC rules were harder to implement than IAC rules. This initial impression is wrong because the separation of transformation and comparison made SAC rules easier to write and validate compared to IAC rules. In terms of implementation size, SAC consists of 3.1 KLOC and IAC of 2.6 KLOC—a rather small difference in size caused primarily by the larger number of rules. It must be noted that our tools are prototypes and were created solely to support the empirical evaluation discussed in this paper. It is our goal to mature the SAC tool and transition it to industry. We have had some success with the abstraction technology already. It should also be noted that the SAC and IAC rules were handcoded. For the comparison between IAC and SAC, this fact is irrelevant. However, for ease of use and transitioning to industry, we plan on extending SAC to provide a better, flexible way for defining transformation and comparison rules.

6 Conclusions

This paper investigated the problem of instant consistency checking between two class models at different levels of abstraction. While traditional approaches to consistency checking typically leveraged from reasonably scalable consistency rules, we found that the consistency checking of class structures was too expensive for instant use. Our original approach, called IAC, implemented consistency rules in the "traditional way" as is often seen in the literature. IAC uses self-contained rules to express the entirety of a consistency concern—containing instructions for the retrieval of all relevant information, their transformation if necessary, and comparison. This integrated transformation and comparison approach to consistency checking works well for many kinds of consistency rules, however, it does not scale where transformation is a computationally expensive task.

Rather than proposing tweaks to deal with this problem, we further investigated this issue and discovered that transformation (abstraction) was fundamentally differently affected by model changes than comparison—and thus, their handling was impaired by integrating them into single rules. We, therefore, developed an alternative approach, called SAC, which separated transformation from comparison. SAC was not only an order of magnitude faster than IAC but also produced fewer false positives and consumed less memory. Our solution, to separate abstraction from comparison, was thus as simple as effective. This paper contributed a scalable and efficient approach to the correct refinement of class models. Our approach, however, is only useful if engineers desire to maintain high and low-level class models separately. The benefit of instant feedback is marginalized if the refinement happens once only and the high-level model is discarded thereafter (i.e., a simple batch consistency mechanism takes longer to compute but if done once only, then this cost is not very significant).

It is worthwhile noting that our approach requires traceability between high- and low-level class models as input. Traceability is currently provided manually. Future work could partially automate its generation analogous to the idea of minimal inconsistency (i.e., what traceability would lead to the least number of inconsistencies). However, the need for traceability is not particular to our approach but generally necessary in cases where models are maintained consistently and separately. Therefore, both IAC and SAC require this input and the explicit need for traceability does not weaken the many benefits of SAC over IAC.

We believe that the separation of transformation and comparison as a paradigm could be applied in other situations, where consistency checking suffers from similar performance related issues.

Acknowledgments

The authors gratefully acknowledge the Austrian Science Fund (FWF) for partially funding this work under agreement P21321-N15.

References

[1] OMG, "Unified Modeling Language Specification Version 1.3," OMG, 1999.

[2] M. Abi-Antoun and N. Medvidovic, "Enabling the Refinement of a Software Architecture into a Design," *Proc. Second Int'l Conf. Unified Modeling Language,* 1999.

[3] M. Moriconi, X. Qian, and R.A. Riemenschneider, "Correct Architecture Refinement," *IEEE Trans. Software Eng.,* vol. 21, no. 4, pp. 356-372, Apr. 1995.

[4] G.C. Murphy, D. Notkin, and K.J. Sullivan, "Software Reflexion Models: Bridging the Gap between Design and Implementation," *IEEE Trans. Software Eng.,* vol. 27, no. 4, pp. 364-380, Apr. 2001.

[5] B.W. Boehm, C. Abts, A.W. Brown, S. Chulani, B.K. Clark, E. Horowitz, R. Madacy, D. Reifer, and B. Steece, *Software Cost Estimation with COCOMO II.* Prentice-Hall, 2000.

[6] R. Balzer, "Tolerating Inconsistency," *Proc. 13th Int'l Conf. Software Eng.,* pp. 158-165, 1991.

[7] S. Fickas, M. Feather, and J. Kramer, *Proc. Int'l Conf. Software Eng. Workshop Living with Inconsistency,* 1997.

[8] S. Easterbrook and B. Nuseibeh, "Using ViewPoints for Inconsistency Management," *IEE Software Eng. J.,* vol. 11, no. 1, pp. 31-43, Jan. 1996.

[9] J. Grundy, J. Hosking, and R. Mugridge, "Inconsistency Management for Multiple-View Software Development Environments," *IEEE Trans. Software Eng.,* vol. 24, no. 11, pp. 960-981, Nov. 1998.

[10] C. Nentwich, L. Capra, W. Emmerich, and A. Finkelstein, "xlinkit: A Consistency Checking and Smart Link Generation Service," *ACM Trans. Internet Technology,* vol. 2, pp. 151-185, 2002.

[11] A. Egyed, "Instant Consistency Checking for the UML," *Proc. Int'l Conf. Software Eng.,* pp. 381-390, 2006.

[12] A. Egyed, "Automated Abstraction of Class Diagrams," *ACM Trans. Software Eng. Methodology,* vol. 11, pp. 449-491, 2002.

[13] A. Egyed, "Compositional and Relational Reasoning during Class Abstraction," *Proc. Sixth Int'l Conf. Unified Modeling Language,* pp. 121-137, 2003.

[14] A. Egyed, "Consistent Adaptation and Evolution of Class Diagrams during Refinement," *Proc. Seventh Int'l Conf. Fundamental Approaches Software Eng.,* pp. 37-53. 2004.

[15] A. Egyed and P. Kruchten, "Rose/Architect: A Tool to Visualize Architecture," *Proc. 32nd Hawaii Int'l Conf. System Sciences,* 1999.

[16] A. Egyed, W. Shen, and K. Wang, "Maintaining Life Perspectives during the Refinement of UML Class Structures," *Proc. Eighth Int'l Conf. Fundamental Approaches to Software Eng.,* pp. 310-325, 2005.

[17] J. Robbins and D. Redmiles, "Cognitive Support, UML Adherence, and XMI Interchange in Argo/UML," *Proc. Int'l Conf. Construction Software Eng. Tools,* pp. 61-70, 1999.

[18] S.L. Pfleeger and S.A. Bohner, "A Framework for Software Maintenance Metrics," *IEEE Trans. Software Eng.,* vol. 16, pp. 320-327, 1990.

[19] K.J. Lieberherr, W.L. Hursch, and C. Xiao, "Object-Extending Class Transformations," *J. Formal Aspects Computing,* vol. 6, pp. 391-416, 1994.

[20] J. Whittle, "Transformations and Software Modeling Languages: Automating Transformations in UML," *Proc. Fifth Int'l Conf. Unified Modeling Language,* pp. 227-242, 2002.

[21] W. Shen and W.L. Low, "Using the Metamodel Mechanism to Support Class Refinement," *Proc. 10th Int'l Conf. Eng. Complex Computer Systems,* pp. 421-430, 2005.

[22] B. Berenbach, "The Evaluation of Large, Complex UML Analysis and Design Model," *Proc. 26th Int'l Conf. Software Eng.,* pp. 232-241, 2004.

[23] A. Tsiolakis and H. Ehrig, "Consistency Analysis of UML Class and Sequence Diagrams Using Attributed Graph Grammars," *Proc. Workshop Graph Transformation Systems,* pp. 77-86, 2000.

[24] G. Engels, R. Heckel, and J.M. Küster, "Rule-Based Specification of Behavioral Consistency Based on the UML Meta-Model," *Proc. Fourth Int'l Conf. Unified Modeling Language, Modeling Languages, Concepts, Tools,* pp. 272-286, 2001.

[25] R. van Der Straeten, T. Mens, J. Simmonds, and V. Jonckers, "Using Description Logic to Maintain Consistency between UML Models," *Proc. Sixth Int'l Conf. Unified Modeling Language,* 2003.

[26] R.H. Bourdeau and B.H.C. Cheng, "A Formal Semantics for Object Model Diagrams," *IEEE Trans. Software Eng.,* vol. 21, no. 10, pp. 799-821, Oct. 1995.

[27] C.A.R. Hoare, *Communicating Sequential Process.* Prentice-Hall, 1985.

[28] J.B. Abrial, *The B-Book.* Cambridge Univ. Press, 1996.

[29] J.M. Spivey, *The Z Notation: A Reference Manual.* Prentice-Hall, 1989.

[30] *The Description Logic Handbook: Theory, Implementation and Applications,* F. Badder, D. Calvanese, D. McGuinness, D. Nardi, and P. Patel-Schneider, eds. Cambridge Univ. Press, 2003.

[31] H. Rasch and H. Wehrheim, "Checking Consistency in UML Diagrams: Classes and State Machines," *Proc. Sixth IFIP WG 6.1 Int'l Conf. Formal Methods Open Object-Based Distributed Systems,* pp. 229-243, 2003.

[32] F.S.E. Ltd., *Failures-Divergence Refinement: FDR2 User Manual,* 1997.

[33] W.L. Yeung, "Checking Consistency between UML Class and State Models Based on CSP and B," *J. Universal Computer Science,* vol. 10, pp. 1540-1559, 2004.

[34] H. Treharne and S. Schneider, "Using a Process Algebra to Control B OPERATIONS," *Proc. First Int'l Conf. Integrated Formal Methods,* pp. 437-457, 1999.

[35] A. Finkelstein, D. Gabbay, A. Hunter, J. Kramer, and B. Nuseibeh, "Inconsistency Handling in Multi-Perspective Specifications," *IEEE Trans. Software Eng.*, vol. 20, pp. 569-578, 1994.

[36] S. Yao and S. Shatz, "Consistency Checking of UML Dynamic Models Based on Petri Net Techniques," *Proc. 15th Int'l Conf. Computing,* pp. 289-297, 2006.

[37] R. Wagner, H. Giese, and U. Nickel, "A Plug-In for Flexible and Incremental Consistency Management," *Proc. Workshop Consistency Problems UML-Based Software Development II,* 2003.

[38] A. Egyed, "Fixing Inconsistencies in UML Design Models," *Proc. Int'l Conf. Software Eng.,* pp. 292-301, 2007.

[39] C. Nentwich, W. Emmerich, and A. Finkelstein, "Consistency Management with Repair Actions," *Proc. 25th Int'l Conf. Software Eng.,* pp. 455-464, 2003.

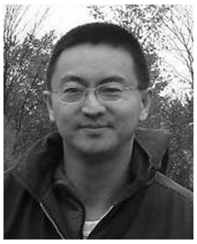
[40] S. Johann and A. Egyed, "Instant and Incremental Transformation of Models," *Proc. 19th Int'l Conf. Automated Software Eng.,* pp. 362-365, 2004.

[41] B. Meyer, *Object-Oriented Software Construction.* Prentice-Hall, 1997.

[42] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1994.

[43] G. Booch, R. Maksimchuk, M. Engle, B. Young, J. Conallen, and K. Houston, *Object-Oriented Analysis and Design with Applications,* third ed. Addison-Wesley, 2007.

Wuwei Shen received the PhD degree in computer science from the Department of Electrical Engineering and Computer Science at the University of Michigan in 2001. He is currently working as an associate professor at Western Michigan University. His research interests include object-oriented analysis and design modeling, model consistency checking, and model-based software testing. He is a member of the IEEE.



Kun Wang received the BS degree in computer science and engineering from Harbin Institute of Technology in 1998 and the MSc degree in computer science from the University of Windsor in 2003. He is working toward the PhD degree in the Department of Computer Science at Western Michigan University. He is now a software engineer at Siemens PLM Software. His primary research interest is software engineering with an emphasis on techniques and tools for improving software quality. His recent work involves detecting inconsistencies in UML-based software development.

Alexander Egyed received the doctorate degree from the University of Southern California in 2000 under the mentorship of Dr. Barry Boehm. He is currently a professor at the Johannes Kepler University, Linz, Austria, where he heads the Institute for Software Engineering and Automation. Previously, he worked as a research scientist at Teknowledge Corporation, and then as a research fellow at University College London, United Kingdom. His research interests include software design modeling, traceability, requirements engineering, consistency checking and resolution, and change impact analysis. He is a member of the IEEE.