Automatically Detecting and Tracking Inconsistencies in Software Design Models

Alexander Egyed, *Member*, *IEEE*

Abstract—Software models typically contain many inconsistencies and consistency checkers help engineers find them. Even if engineers are willing to tolerate inconsistencies, they are better off knowing about their existence to avoid follow-on errors and unnecessary rework. However, current approaches do not detect or track inconsistencies fast enough. This paper presents an automated approach for detecting and tracking inconsistencies in real time (while the model changes). Engineers only need to define consistency rules—in any language—and our approach automatically identifies how model changes affect these consistency rules. It does this by observing the behavior of consistency rules to understand how they affect the model. The approach is quick, correct, scalable, fully automated, and easy to use as it does not require any special skills from the engineers using it. We evaluated the approach on 34 models with model sizes of up to 162,237 model elements and 24 types of consistency rules. Our empirical evaluation shows that our approach requires only 1.4 ms to reevaluate the consistency of the model after a change (on average); its performance is not noticeably affected by the model size and common consistency rules but only by the number of consistency rules, at the expense of a quite acceptable, linearly increasing memory consumption.

Index Terms—Design tools and techniques, design.

1 Introduction

Large design models contain thousands of model elements. Engineers easily get overwhelmed maintaining the consistency of such design models over time. Not only is it hard to detect new inconsistencies while the model changes but it is also hard to keep track of known inconsistencies. To date, many consistency checking mechanisms exist, but most of them are only capable of checking the consistency of design models as a whole (in a batch process), where all consistency rules are evaluated on the entire model [3], [8], [37]. Unfortunately, batch consistency checking does not scale and the checking of larger models takes hours to complete. As a result, engineers only use such batch consistency checkers occasionally—waiting days, perhaps even weeks, before checking the consistency of a model—at which time they are overwhelmed with many inconsistencies (the worst industrial model we investigated contained 10,466 inconsistencies). To fix these inconsistencies, engineers then have to interrupt their workflow further to reinvestigate the model changes that led to these inconsistencies—model changes made hours, days, or even weeks earlier. Engineers then not only have to fix the erroneous model changes that led to the inconsistencies but they also have to correct follow-on decisions (i.e., other model changes) that were based on the erroneous model elements [5]. Batch consistency checking thus cannot keep up with the engineers, provides late feedback, and interrupts the workflow of the engineers involved.

• *The author is with the Johannes Kepler University, Altenbergerstr. 69, 4040 Linz, Austria. E-mail: alexander.egyed@jku.at.*

Manuscript received 14 July 2009; revised 14 Dec. 2009; accepted 28 Dec. 2009; published online 2 Mar. 2010.
Recommended for acceptance by B. Nuseibeh.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSE-2009-07-0178.
Digital Object Identifier no. 10.1109/TSE.2010.38.

Instant feedback of any kind is a fundamental best practice in the software engineering process. Today, programmers benefit from instant compilation and integrated development environments (IDEs) point out many (if not all) syntax and semantic errors within seconds of making them—usually in a nonintrusive manner. Although there are several modeling tools that support the incremental consistency checking of design models, they either do not provide design feedback instantly [27] or they require extensive manual overhead in annotating or rewriting consistency rules [4], [31]. This is because correctly deciding how model changes affect consistency rules is a complex task given the very large number of potential changes involved. Any manual overhead in deciding this is bound to be error prone.

This paper presents an automated approach to the incremental consistency checking of software design models. We demonstrate that our approach keeps up with an engineer's rate of model changes, even on very large industrial models with tens of thousands of model elements. Our approach fully automatically, correctly, and efficiently decides what consistency rules to evaluate when the model changes. It does so by observing the behavior of consistency rules during validation (i.e., what model elements were accessed during the evaluation of a rule). To this end, we developed the equivalent of a model profiler for consistency checking. The profiling data are used to establish a correlation between model elements and consistency rules (and inconsistencies). Based on this correlation, we decide when to reevaluate consistency rules and when to display inconsistencies—allowing an engineer to quickly identify all inconsistencies that pertain to any part of the model of interest at any time (i.e., living with inconsistencies [2], [17]). Our approach treats consistency rules as black boxes. Consistency rules neither have to be written in a special language nor do they have to be annotated in any way. This independence of the constraint language is very important because we found that engineers are neither capable of nor willing to use special languages and/or annotations. This is then a particularly severe problem when engineers create their own variations of nonstandard consistency rules, as is often done in UML [33]. **Engineers can thus define consistency rules at will, in any constraint or modeling language.** Our approach is also integrated into the modeling tool IBM Rational Software Modeler for ease of use and broader applicability.

Even though our approach (or any approach) is not guaranteed to be instant for every consistency rule (i.e., consistency rules could be written globally without being able to break them down locally), this paper presents empirical evidence that our approach is easily able to keep up with an engineer's rate of model changes for the 24 consistency rules we studied. These consistency rules cover the most significant concerns of keeping sequence diagrams consistent with class and statechart diagrams. Our approach evaluated these rules on 34 UML design models totaling over 280,000 model elements. Our empirical data show that the evaluation of a model after changes averaged to less than 1.4 ms (in only 0.00011 percent of changes was the evaluation time $>$ 100 ms but never worse than 2 seconds)—even on the largest industrial models we had available. This benefit comes at the expense of a linearly increasing memory cost to store the observed behavior of consistency rules. Our approach can be used to provide consistency feedback in an intrusive or nonintrusive manner. It may also be coupled with inconsistency actions to resolve errors automatically [13], [15], [28]. However, space limitations preclude these discussions.

To date, our technology has been applied to UML 1.3, UML 2.1, Matlab/Stateflow, and the Dopler product line to demonstrate our approach's applicability to different modeling notations. We also implemented consistency checkers and rules using a range of different languages—Java, C#, J#, and Object Constraint Language (OCL) [38])—to demonstrate our approach's applicability to diverse constraint languages. Indeed, we believe that our approach supports any modeling or consistency language for as long as it is possible to observe the constraints evaluation, as is discussed later. An earlier version of this paper implementing UML 1.3 only and evaluated on 29 design models appeared in [12].

2 Problem

The following section describes consistency rules and outlines the problem of how to evaluate them incrementally. The discussion in this paper is accompanied by a simple model illustration.

2.1 Consistency Rules and Illustration

The illustration in Fig. 1 depicts three diagrams created with the UML [17] modeling tool IBM Rational Software Modeler. The given model represents an early design-time snapshot of a video-on-demand (VOD) system [4]. The class diagram (top) represents the structure of the VOD system: a *Display* used for visualizing movies and receiving user input, a *Streamer* for downloading and decoding movie streams, and a *Server* for providing the movie data. In UML, a class's behavior can be described in the form of a statechart diagram. We did so for the *Streamer* class (middle). The behavior of the *Streamer* is quite trivial. It first establishes a connection to the server and then toggles between the *waiting* and *streaming* mode depending on whether it receives the *wait* and *stream* commands.

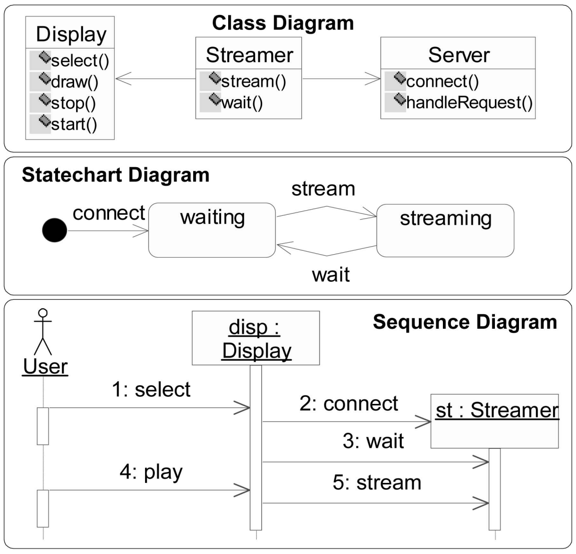


Fig. 1. Simplified UML model of the VOD system.

The sequence diagram (bottom) describes the process of selecting a movie and playing it. Since a sequence diagram contains interactions among instances of classes (objects), the illustration depicts a particular user invoking the *select* method on an object, called *disp*, of type *Display*. This object then creates a new object, called *st*, of type *Streamer*, invokes *connect* and then *wait*. When the user invokes *play*, object *disp* invokes *stream* on object *st*.

These UML consistency rules describe conditions that a UML model must satisfy for it to be considered a valid UML model. Fig. 2 lists 24 such rules covering consistency, well-formedness, and best practice criteria among UML class, sequence, and statechart diagrams. The first four consistency rules are elaborated on for better understanding. *Note that these consistency rules apply to UML only. For the other modeling notations, different consistency rules were needed, which are not described here.*

A consistency rule may be thought of as a *condition* that evaluates a portion of a model to a *truth value* (true or false). For example, consistency rule 1 states that the name of a message must match an operation in the receiver's class. If this rule is evaluated on the third message in the sequence diagram (the *wait* message), then the condition first computes *operations = message.receiver.base.operations*, where *message.receiver* is the object *st* (this object is on the receiving end of the message; see arrowhead), *receiver.base* is the class *Streamer* (object *st* is an instance of class *Streamer*), and *base.operations* is {*stream(),wait()*} (the list of operations of the class *Streamer*). The condition then returns true because the set of operation names (*operations> name*) contains the message name *wait*.

Rule	Description and Implementation

1	**Name of message must match an operation in receiver's class**
	operations=message.receiver.base.operations & base.parents.operations
	return operations->name->contains(message.name)

2	**Calling direction of message must match an association**
	in=message.receiver.base.incomingAssociations & base.parents.incomingAssociations;
	out=message.sender.base.outgoingAssociations & base.parents.outgoingAssociations;
	return in.intersectsWith(out)

3	**Sequence of object messages must correspond to events** startingPoints = find state transitions equal first message name startingPoints->exists(message sequence equal transition sequence reachable from startingPoint)
4	**Cardinality of association must match sequence interaction**
5	**Statechart action must be defined as an operation in owner's class**
6	**Parent class attribute should not refer to child class**
7	**Parent class should not have a method with a parameter referring to a child class**
8	**Association ends must have a unique name within the association**
9	**At most one association end may be an aggregation or composition**
10	**The connected classifiers of the association end should be included in the namespace of the association**
11	**The class of an association end cannot be an interface if there is an association navigable away from that end**
12	**A classifier may not belong by composition to more than one composite classifier**
13	**Method parameters must have unique names**
14	**Type of Method Parameters must be included in the Namespace of method owner**
15	**A class may not use the same attribute names as outgoing association end names**
16	**No two behavioral features may have the same signature in a classifier**
17	**No two attributes may have the same name within a class**
18	**A classifier may not declare an attributes that has been declared in parents**
19	**Outgoing association ends names must be unique within classifier**
20	**The elements owned by a namespace must have unique names**
21	**An interface can only contain public operations (no attributes)**
22	**No circular inheritance**
23	**A generalizable element may only be a child of another such element of the same kind**
24	**The parent must be included in the Namespace of the GeneralizableElement**

Fig. 2. Consistency rules for UML class, sequence, and statechart diagrams. Details sketched for first three rules only. Rules 7 and 8 are classical best practice rules (and not necessarily errors). Rules 9-25 are typical UML well-formedness rules defined in UML 1.3. Different rules apply to other modeling languages (e.g., Dopler).

The model also contains inconsistencies. For example, there is no *connect()* method in the *Streamer* class although the *disp* object invokes *connect* on the *st* object (rule 1). The *disp* object calls the *st* object even though, in the class diagram only, a *Streamer* may call a *Display* (rule 2). Or, the sequence of incoming messages of the *st* object ($connect > wait > stream$) is not supported by the statechart diagram, which expects a *stream* after a *connect* (rule 3).

It is generally true that consistency rules are stateless and deterministic. Our approach certainly presumes this. That is, if any rule is evaluated on the same portion of the model twice, then it will perform the same actions (i.e., access the same model elements in the same order) and determine the same truth value. In the following, we define a **model element** to be an instance of a metamodel element. For example, all messages (e.g., *wait*) are instances of the UML metamodel element *Message*.

Consistency rules are typically evaluated from the viewpoint of a metamodel element to ease their design and maintenance—the so-called **context element**. For example, consistency rule 1 is expressed from the view of a UML Message (i.e., given a message, is it consistent?). The message is thus the context element of Rule 1. It is common practice to define consistency rules with context elements. Even commercial modeling tools, such as IBM Rational Software Modeler, do so:

$$\begin{aligned}&\mathit{ConsistencyRule} =\\ &\quad <\mathit{Condition}, \mathit{ContextElement}> \rightarrow \mathit{Bool}\\ &\quad \mathit{where}\ \mathit{ContextElement} \in \mathit{MetaModelElements}.\end{aligned}$$

Consistency rules are certainly affected by changes to their context elements; however, it is important to understand that consistency rules are also affected by many other model elements not explicitly identified. **The most complex problem of incremental consistency checking is thus in correctly finding all model elements that affect the truth value of any given consistency rule.**

2.2 Understanding Changes

Since consistency rules are conditions on a model, their truth values change only if the model changes (or if the condition changes, but this is explored elsewhere [20]). Instant consistency checking thus requires understanding *when*, *where*, and *how* the model changes. However, changes to models are not simple events. Typically, a single user change implies a sequence of model changes. For example, if an engineer creates a message between two objects in a sequence diagram, then this change causes the creation of a new message (a new model element) and it modifies two existing objects: the objects in the sequence diagram:

- New model element of type *Message* with ID 1.
- Modified model element of type *Object* [*outgoingMessages*] with ID 2.
- Modified model element of type *Object* [*incomingMessages*] with ID 3.

The first change notification tells about the creation of a model element—an instance of a UML *Message* with an id that uniquely identifies the model element. Since a message was created between two existing UML Objects, these objects are modified in that one now owns a new incoming message and the other a new outgoing message.

It is important to note that model elements are aggregations of fields. For example, a message has a *name* field (of type *string*) or a *receiver* field (a reference to the UML *Object*) to describe the properties of a message and its relationship to other model elements. In the case of the UML, the metamodel describes them in detail. For consistency checking, it is important to note that **changes typically modify single fields of a model element only (except for the creation or deletion of model elements)**. For example, the creation of the message only changed the *outgoingMessages* and *incomingMessages* fields of both objects (not modified were, say, the *name* fields of both objects):

$$\textit{Change} = <\textit{ModelElement}, \textit{Field}>: \textit{oldValue} \rightarrow \textit{newValue},$$
$$\textit{ChangeGroup} = \textit{Set of} <\textit{Change}>.$$

It is obvious that a new message must be created before it can be added between the two objects. Yet there is no obvious ordering on whether the message is added to the *outgoingMessages* field of the object first or last. Changes are thus grouped together to ensure well-formedness. This distinction between changes and change groups will become important later because the impact of a change is in fact the impact of the entire change group and incremental consistency checking on a model only makes sense if an entire change group is considered.

2.3 Understanding Impact of a Change

It is intuitive to think of instant consistency checking in terms of *what happens if* a model element changes [4]. For example, we know that the message *wait* in the sequence diagram is consistent with respect to rule 1 (i.e., the *Streamer* class has a method with the same name). This truth is violated if the engineer changes the name of the message, say, from *wait* to *suspend* (i.e., the *Streamer* class does not have the method *suspend*). A change to a message name thus requires the reevaluation of the consistency rule 1.

However, a change to the message name is not the only way the message *wait* can be made inconsistent with respect to rule 1. For example, if the engineer renames the class

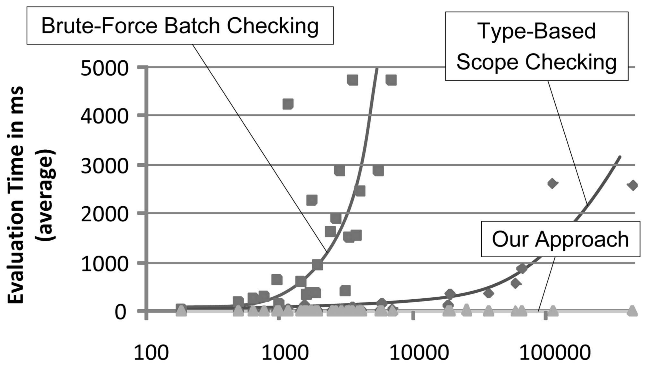


Fig. 3. Evaluation time of a model change.

method *wait* into *suspend*, then there is no longer a method that matches the message name. This change also invalidates rule 1. And there are several other changes like that. Likewise, there are many changes that do not affect consistency rule 1. For example, a change to the directionality of the associations (arrows) between the classes will never affect consistency rule 1. Given the many ways on how model changes affect consistency rules (or do not affect them), it is difficult to identify them all manually (as is required by most mainstream design tools).

Most mainstream design tools (ArgoUML [15], IBM Rational Software Modeler) provide mechanisms for identifying what *types of model changes* account for what *types of inconsistencies* (i.e., a change to a message name may violate consistency rule 1). These approaches rely on what we refer to as a type-based scope for incremental consistency checking—a coarse-grained filter that improves performance but does not eliminate the basic scalability problem. Fig. 3 depicts the scalability problem on 34 sample models. These mostly industrial models are discussed in more detail in Section 5. However, we see that the models we used span a wide range of model sizes, with the largest having 162,237 model elements (and over 435,932 fields). It shows that batch consistency checking (that evaluates all consistency rules) becomes expensive quickly. Approaches such as ArgoUML, which use type-based scopes, fare significantly better. Yet we see that even such approaches do not scale well and are no longer instant for medium-sized models. It must also be stressed that these values are based on 24 consistency rules only. The computational cost further increases with the increasing number of consistency rules involved. The performance of a type-based approach to consistency checking is thus far from ideal—and certainly not instant. Considering that the type-based scope must be computed manually, there is also no guarantee of correctness.

Fig. 3 also shows the performance of our approach, discussed next. We see that our approach is much faster, without any noticeable scalability effects in terms of model size, and does not require manual annotations for consistency rules (like the manually created type-based scope). We see that the average computational cost of a model change is in the range of milliseconds, even for the largest models we had available.

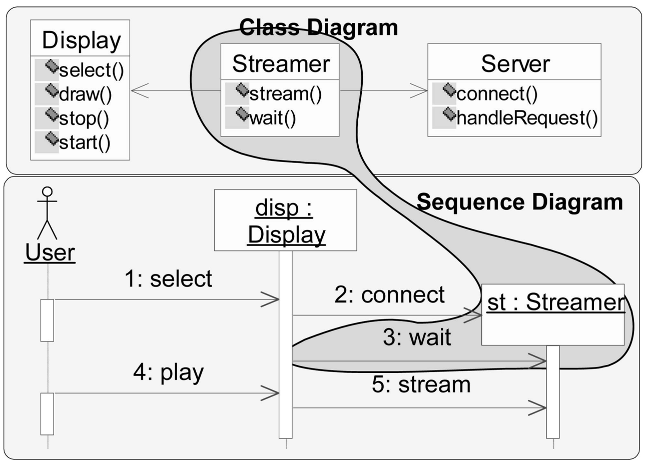


Fig. 4. Scope for message *Wait* evaluated by consistency rule 1 (short <Rule 1, wait>).

3 Instance-Based Scope to Consistency

To improve on the performance of type-based consistency checking, we work with the actual model elements—the instances of UML metamodel elements. To support the fast, incremental checking of design changes, our approach identifies how changes to model elements (indeed, to their individual fields) affect the truth values of consistency rules. A consistency rule needs to be reevaluated if one such model element changes. We refer to the set of model elements that affect the truth value as the **change impact scope** of a consistency rule—or *scope*, for short. The scope must be complete for our approach to be correct and the scope must be small for our approach to be efficient. We will demonstrate that our approach produces complete and small (albeit not minimal) scopes. The concept of a scope is simple in principle; however, thus far nobody has been able to compute it in advance. Without this computation, we do not know what consistency rules to evaluate when the model changes. Our approach circumvents this problem by observing the runtime behavior of consistency rules during their evaluation. To this end, we developed the equivalent of a *model profiler* for consistency checking.

In the following, we first discuss the benefits of treating every evaluation of a consistency rule as a separate event—we speak of consistency rule instances (CRIs). Next, we discuss the fact that the set of model elements accessed during a CRI's evaluation is in fact a superset of the minimal change impact scope.

3.1 Consistency Rules and Their Instances

During evaluation, a consistency rule requires access to a portion of the model only. Recall that the evaluation of consistency rule 1 on message *wait* first accessed the message *wait*, then the message's receiver object *st*, next its base class *Streamer*, and finally, the methods *stream()* and *wait()* of the base class (Section 2.1). This behavior was defined in Fig. 2. Rule 1 evaluated on message *wait* thus accesses the model elements {*wait*, *st*, *Streamer*, *stream()*, *wait()*}, as illustrated in Fig. 4. We will demonstrate later that the minimal, complete change impact scope is a subset of these accessed model elements.

Recall from Section 2.1 that a consistency rule is typically written from the viewpoint of a context element (a type of model element) from where its evaluation starts. For consistency rule 1, the context element is a UML Message. Since there are five such messages in the sequence diagram in Fig. 4, consistency rule 1 must be evaluated five times—once for every message. To distinguish these evaluations, we define a CRI to represent each such evaluation (<consistency rule, model element> pairs):

$$\begin{aligned}CRI &= <ConsistencyRule, Model\ Element>, where\\ &ModelElement\ instance\ of\ ContextElement\\ &(ConsistencyRule).\end{aligned}$$

Every CRI starts its evaluation at a different instance of the context element (e.g., at different messages in case of consistency rule 1). Every CRI accesses different model elements (though the set of accessed model elements may overlap), and consequently may return different truth values (e.g., the evaluation of message *play* fails, whereas the evaluation of message *wait* succeeds). Not surprisingly, **each CRI is affected differently by model changes, which is why our approach identifies the change impact scope for each CRI separately**.

For example, from above, we know that the evaluation of consistency rule 1 on message *wait* (short <rule1, wait>) accesses the model elements {*wait*, *st*, *Streamer*, *stream()*, *wait()*}. Yet, the evaluation of consistency rule 1 on message *play* (short <rule1, play>) requires access to {*play*, *disp*, *Display*, *select()*, *draw()*, *stop()*, *start()*}. The model elements accessed by <rule1, play> are different from the ones accessed by <rule 1, wait> even though both evaluations are based on the same consistency rule.

Our observation is that the evaluation of a CRI accesses *at least* those model elements that are needed to determine its truth value. Since we presume consistency rules to be stateless and deterministic (recall Section 2.1), it follows that solely these accessed model elements are needed to compute their truth values. Or, on the contrary, a model element that was not accessed during a CRI's evaluation cannot contribute to its truth value. If a model element changes, then all of those CRIs have to be reevaluated that accessed the changed model element. For example, if method *wait* is renamed to a nonexisting method name such as *foo*, then CRI <rule1, wait> needs to be evaluated, whereas CRI <rule1, play> need not (because it did not previously access the changed method name). We thus define the **scope** of a CRI to be the elements accessed during its evaluation. And if a model element changes, then only those CRIs are affected (and must be reevaluated) that include the changed element in their respective scopes (we discuss this further in Section 3.3):

$$\begin{aligned}Scope(CRI) &= Set\ of <ModelElement, Field> pairs\\ &\quad accessed\ during\ Evaluation\ of\ CRI,\\ AffectedCRIs(Change) &=\\ &\{CRI \in CRIs/CRI.ScopecontainsChange\}.\end{aligned}$$

It is important to note that our change impact scopes are in fact quite small (a fact that will be supported through extensive empirical evidence in Section 5). It is also important to note that consistency rules typically access a few selected fields of model elements only and it is quite possible that two consistency rules access some of the same model elements, but different fields thereof (for example, rule 1 accesses the operations of a class whereas rule 2 accesses the associations of a class—two distinct fields). Since fields of a model element can be changed separately, it follows that scopes should be maintained as model element/field pairs rather than model elements—another highly significant performance benefit. The precise list of <model element, field> pairs accessed during the evaluation of <rule 1, *wait*> is thus:

- Message *wait* [name]: value = String "wait."
- Message *wait* [receiver]: value = Object *st*.
- Object st [base]: value = Class *Streamer*.
- Class *Streamer* [operations]: value = Set of Methods {stream(), wait()}.
- Method *stream()* [name]: value = String "stream."
- Method *wait()* [name]: value = String "wait."

3.2 Scope Detection and Correctness

Our approach requires a complete change impact scope but not necessarily a minimal scope for correctness. Any missing model element in the scope would be problematic because the approach would not reevaluate all CRIs affected by changes. To the best of our knowledge, it is not possible to compute the scope of CRIs automatically by statically analyzing consistency rules (i.e., through formal analysis). Some prediction models exist that evaluate the impact of a change [25]. Some even tried to define explicit change impact rules that complement consistency rules in identifying when and how changes impact a model [6], [24]. Yet others simply rewrite consistency rules multiple times to account for different kinds of changes [4]. However, all of these approaches require extensive manual effort and are not guaranteed to be correct.

We, on the other hand, automatically compute the scopes of CRIs by observing their behavior through model profiling. Profilers are used extensively on source code to observe what code is executed when and in what order during the execution of a software system. Our model profiler is similar in that it observes what model elements are accessed when and in what order *during consistency checking*. The infrastructure for modeling our profiling technology is discussed in [14]. Through the help of the model profiler, it is simple to detect the scope for any given CRI by letting the profiler log all <model elements, field> pairs accessed during the evaluation of a CRI and storing the accessed model elements in its scope thereafter. This scope is observable fully automatically and we will demonstrate that this scope only changes when the CRI needs to reevaluate.

Next, we investigate whether 1) the scope is complete, 2) the scope is small, and 3) how the scope is affected by changes. The third issue is particularly important because the change impact scope of a CRI does not stay constant over time. We first investigate these questions for existing CRIs. Section 3.3 then investigates how to create and destroy CRIs in response to model changes. That is, it is one problem to correctly reevaluate existing CRIs and it is another problem to maintain CRIs.

3.2.1 The Scope Is Complete

The correctness of our approach requires the scope to include at least those model elements that affect its truth value. Fortunately, one may err in favor of having more elements in the scope than needed, causing potentially unnecessary evaluations (reduced performance) but not omitting necessary ones. Our premise is that consistency rules are stateless and deterministic (recall Section 2.1). The same rules invoked on the same model use the exact same model elements and result in the exact same truth values time and time again. Thus, the scope inferred through a rule's evaluation is deterministic, repeatable, and includes all model elements required to determine the truth value.

The completeness issue is obvious for simple, unconditional statements, where a consistency rule navigates a set of model elements. For example, consistency rule 1 first identifies all of the methods in the path *message.receiver.base.operations*. There, the consistency rule accesses the set of model elements based on a relative path (i.e., starting from some message, access its receiver, and so on). All elements along such paths will become part of the scope.

Profiling makes this easy because the path corresponds to a sequence of interactions with the model. For example, the path *message.receiver.base.operations* starts at a particular message, obtains its receiver by invoking *getReceiver()* on that message, which returns an object (the receiver of a message is an object). The profiler thus records that the *receiver* field of that message was accessed (note that message and object are concrete elements: For example, for message *wait* in Fig. 4, the *getReceiver()* returns the object *st*). The path next accesses the *base* of that object by invoking *getBase()*—which returns a class (class *Streamer* in case of message *wait*). The profiler thus records that the *base* field of that object was accessed. The path then accesses the *operations* of that class by invoking *getOperations()*—which returns a collection of operations. The profiler thus records that the *operations* field of that class was accessed. The operations will eventually be added to the scope because the subsequent existence check will access the names of these operations until one is found that matches the method name. Thus, the profiler also records that the *name* field of those operations was accessed and the name field of the message.

Consistency rules rely on path expressions for locating model elements but they also rely on a range of first-order logic for further processing. Fig. 5 lists common constructs found in consistency rules [28] and it also points out that there are many constructs where model elements are potentially accessible but not accessed during the evaluation of a consistency rule. We will see next that our completeness property holds despite the fact that not all model elements are captured in the scopes that are accessible by a consistency rule.

Take, for example, the OR operator. The OR operator requires a condition to be evaluated only until the first subcondition is true. For example, if A is true in A *or* B, then B is not evaluated. Only if A is false is B then evaluated. Thus, if A is true in A *or* B, then only A is accessed and added to the scope but B is not. It follows that B is not in the scope and the scope thus appears incomplete. Fortunately, this level of completeness is not required for our

Result	**Consistency Rule Statements**	**Accessed Model Elements**
path	sequence of elements	all elements in scope
	element	element in scope
	element1.element2	element1 and element2 in scope
condition	function(path1,path2)	elements in path1 and path2 in scope (functions: =, <, >,…)
condition	function(path1)	elements in path1 in scope (functions: size, count,…)
condition	condition1 implies condition2	elements in condition1 in scope always elements in condition2 in scope only if condition1=true
condition	condition1 and condition2	elements in condition1 in scope always elements in condition2 in scope only if condition1=true
condition	condition1 or condition2	elements in condition1 in scope always elements in condition2 in scope only if condition1=false
condition	not condition1	elements in condition1 in scope
condition	for all elements in path: condition	all elements in path in scope only if condition=true applies to all elements; otherwise subset of elements in path in scope
condition	exists element in path: condition	all elements in path in scope only if condition=false applies to all elements; otherwise subset of elements in path in scope

Fig. 5. Accessible and accessed model elements.

problem. We only require the scope to include those elements that affect a consistency rule's truth value. If A is true in A *or* B, then A is the only element contributing to the OR expression's truth value. In other words, for as long as A remains true in A *or* B, changes to B do not matter and B is thus not required to be in the scope.

In addition to the OR constructs, Fig. 5 lists other constructs where not all accessible model elements are accessed (e.g., and, for all, exists quantifications, implies) but there completeness also only requires accessed elements. The scope determined by our approach is thus complete because it contains at least the model elements needed for its evaluation.

3.2.2 The Scope Is Not Minimal but Small and Bounded

A minimal scope guarantees that a rule is evaluated only if its truth value changes. Any reevaluation that does not change a consistency rule's truth value is unnecessary because it recomputes what is already known. We believe that it is infeasible to eliminate all unnecessary evaluation without introducing manual and error-prone change annotations. Yet, we have to be careful in limiting the scope, i.e., bounding it to some maximum size. Our approach has this upper bound in scope size: We already know that a rule's evaluation uses at most all potentially accessible model elements. Our scope is thus bounded to not include model elements that do not affect the truth value of a consistency rule. We evaluated whether this bounded scope is enough to ensure computational scalability and Section 5 presents the empirical evidence that the scope sizes, while not minimal, were small in including 21 model elements or fewer for 95 percent of all CRIs. But most significantly, we found that the scope sizes did not increase with model sizes. They stayed constant.

This is not to say that the engineers or tool builders have no role in deciding on the efficiency of consistency checking. The implementation of consistency rules very much affects the efficiency of consistency checking, the scope sizes, and thus the computational cost of incrementally reevaluating the consistency of a model change. To illustrate this, consider the two different, though equivalent, implementation choices for consistency rule 1 in Fig. 6. The top implementation identifies all operations in the receiver's base class and all of its parent classes. It then retrieves the names of all of these operations and checks whether the message name is contained among the operation names. The second implementation retrieves the base class only and first iterates over its methods before proceeding to the parent classes. The difference in terms of scope is that the first implementation accesses all names of all operations (base and parents) whereas the second implementation searches incrementally from base class to parent classes until it finds a suitable method. The second implementation is computationally more efficient to evaluate and it has a smaller scope than the first implementation, which also causes it to be reevaluated less often (smaller scope implies fewer reevaluations).

Since both implementations are equivalent (i.e., they correctly implement consistency rule 1), how are we to

Implementation Choice 1 for Consistency Rule 1:

```
operations=msg.receiver.base.operations &
           msg.receiver.parents.operations
return operations->name->contains(message.name)
```

Implementation Choice 2 for Consistency Rule 1:

```
classes=msg.receiver.base
while classes is not empty
  for each operation in classes->operations
    if (operation.name equals msg.name) return true
  end for
  classes =classes->parents
end while
return false
```

Fig. 6. Two behaviorally different implementations of consistency rule 1 that are equivalent in terms of their results.

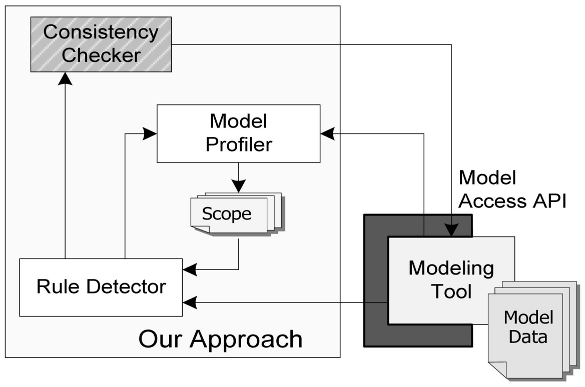


Fig. 7. Architecture of model/analyzer.

interpret their differences in scope? The answer is similar to the discussion on the OR construct above. The first implementation accesses model elements even though it does not necessarily need them (i.e., if the base class has a suitable method, then accessing the parent classes is unnecessary). The second implementation accesses model elements only if they are needed. Therefore, the first implementation includes unnecessary model elements in its scope but both implementations include all necessary change impact scope elements. Both implementations are valid and our approach would correctly handle both—albeit the second more efficient than the first one.

3.2.3 The Scope Changes over Time

The scope of a CRI is not constant but changes over its life. For example, the scope for CRI <*rule1*, *wait*> is {*wait*, *st*, *Streamer*, *stream()*, *wait()*} in Fig. 1. If the base of object *st* changes from *Streamer* to, say, *Server*, then the evaluation of the same CRI <*rule1*, *wait*> accesses *Server* (the new base class) but not *Streamer* (the old base class). Thus, after the change, the scope of <*rule1*, *wait*> is {*wait*, *st*, *Server*, *connect()*, *handleRequest()*}. The class *Streamer* and its methods no longer affect this CRI's truth value.

Fortunately, the scope of a CRI changes only if a model element in its scope changes—triggering the CRI's reevaluation. Thus, the recomputation of a rule coincides with the reevaluation of its truth value. This implies that the scope of CRIs must be recomputed every time the CRI is reevaluated. The overhead of this recomputation is negligible.

3.2.4 Creating and Destroying Rule Instances

Instant consistency checking requires an understanding of when, where, and how the model changes. For this purpose, our approach monitors the engineer while using the modeling tool. Fig. 7 shows the architecture of our approach. It depicts the modeling tool in the lower right corner. The modeling tool needs to be wrapped such that we can observe user activities—changes to the design model which is embedded inside the modeling tool (i.e., a UML design model).

The *Consistency Checker* (top left) accesses the design model and, while doing so, the *Model Profiler* (middle) monitors what model elements the consistency checker accesses. The profiler then logs the accessed model elements in a scope database, together with the knowledge of what CRIs accessed them. The latter information comes from the *Rule Detector* (bottom left). The rule detector instructs the consistency checker on what CRIs to evaluate. It determines this by observing model changes and looking up what CRIs previously accessed the changed model elements. A simple lookup table is sufficient to locate all affected CRIs for any given changed model element.

There is an obvious chicken-and-the-egg problem here. The rule detector requires the scope database to determine what rules to reevaluate with model changes. This scope database is, however, created *after* the evaluation of the CRIs. There are two alternative solutions for this problem: 1) Force an initial evaluation of all CRIs on model load or 2) save the scope persistently. However, a challenge in both cases is how to know what CRIs should exist. CRIs are continuously created and destroyed during the entire life cycle of a model to reflect the needs of the model. An incremental consistency checker thus needs to keep track of CRIs—it needs to know when and how to create and destroy CRIs. This issue is explored next.

As was briefly discussed in Section 3.1, CRIs live and die with their context elements. Recall that the context element is the starting point for evaluating a consistency rule. In the case of consistency rule 1, the context element was a UML message. If there is no message, then there is no need to evaluate consistency rule 1. If there are multiple messages (as in Fig. 1), then consistency rule 1 must be evaluated multiple times (once per message).

Our approach simply creates CRIs when context elements are created and it destroys CRIs once their context elements are destroyed. Implicitly, this implies that consistency rules must come with the knowledge of what context element they apply to. This was discussed in Section 2.1 when we said that consistency rules are typically evaluated from the viewpoint of a metamodel element. However, note that the context for a consistency rule is defined for a metamodel element and not its instances (e.g., model elements). For example, consistency rule 1 is defined to apply to a UML Message. Any creation of a model element that instantiates a UML Message thus triggers the creation of a CRI. Any destruction of such a model element thus triggers the destruction of a CRI.

The *RuleDetector* algorithm below illustrates rule creation, destruction, and reevaluation in response to model changes. We see that if a model element is created, then all those consistency rules must be instantiated that have a context element equal to the type of the changed element. These new CRIs must be immediately evaluated to compute their truth values and scopes. If a model element is destroyed, then all those CRIs must be destroyed where the context element equals the destroyed element (note: the instances must match). A destroyed CRI need not be evaluated any longer; its scope can be discarded. Independent of rule creation and destruction, the *RuleDetector* algorithm must also process the CRIs affected by the change based on the scope database as was discussed above.

```
RuleDetector(changedElement, scope)
if changedElement was created
   for every rule where
       type(rule.rootElement) = type(changedElement)
```

```
            ruleInstance = new <rule, changedElement>
            evaluate ruleInstance
    end for
else if changedElement was deleted
    for every CRI where
            CRI.contextElement = changedElement
            destroy CRI
    end for
end if
for every CRI where CRI.scope contains changedElement
    evaluate CRI
end for.
```

The life of a CRI is tied to the life of its context element. Moreover, the context element remains constant for any given CRI throughout its life. It is interesting to observe that the creation of CRIs is based on the metamodel elements (e.g., UML Message) whereas the evaluation and destruction of CRIs are based on model elements (e.g., message *connect*). The algorithm above treats the evaluation (bottom) separately from the rule creation and destruction (top). This is because the deletion of a model element could trigger both the destruction of some CRIs and the evaluation of others.

3.3 Processing Entire Change Groups

The *RuleDetector* algorithm shown above is not optimal since it causes certain CRIs to be reevaluated more often than needed. We discussed in Section 2.2 that model changes almost never result in single change notifications because a model change typically affects multiple model elements. We referred to related changes as change groups. Instant consistency checking should recognize these logical groupings among change notifications because the consistency rules are not expected to apply in between changes of a change group but only thereafter. It is not meaningful to check the consistency of a model that is in the middle of a change—say, where a dependency has been removed from one class but not yet added to the other. Furthermore, it is also not useful to reevaluate a CRI more than once per change group.

The *RuleDetector* algorithm discussed above investigated changes instantly and independent of one another. It thus triggers multiple reevaluations of the same CRI in cases where multiple, logically connected changes affect the same CRI. To prevent our approach from evaluating CRIs unnecessarily, our approach first processes all change notifications before evaluating any CRIs to eliminate duplicates. Fig. 8 depicts the internals of the *RuleDetector* component.

A secondary benefit of separating the selection of rules from their evaluation is in potentially delaying consistency checking—from instant consistency checking to lazy consistency checking, if so desired. For example, an engineer may wish to make a sequence of changes before reevaluating the consistency of these changes. This is then particularly useful when changes are explorative and the engineer knows that the changes are likely to conflict with the rest of the model. In such cases, the engineer may not care to receive intermittent inconsistency feedback but rather desires feedback at the end, after the sequence of changes is completed.

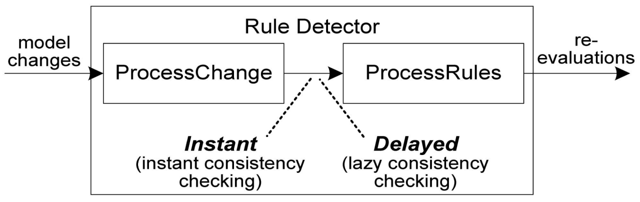

Rule Detector
model changes
ProcessChange
ProcessRules
re-evaluations
Instant (instant consistency checking)
Delayed (lazy consistency checking)

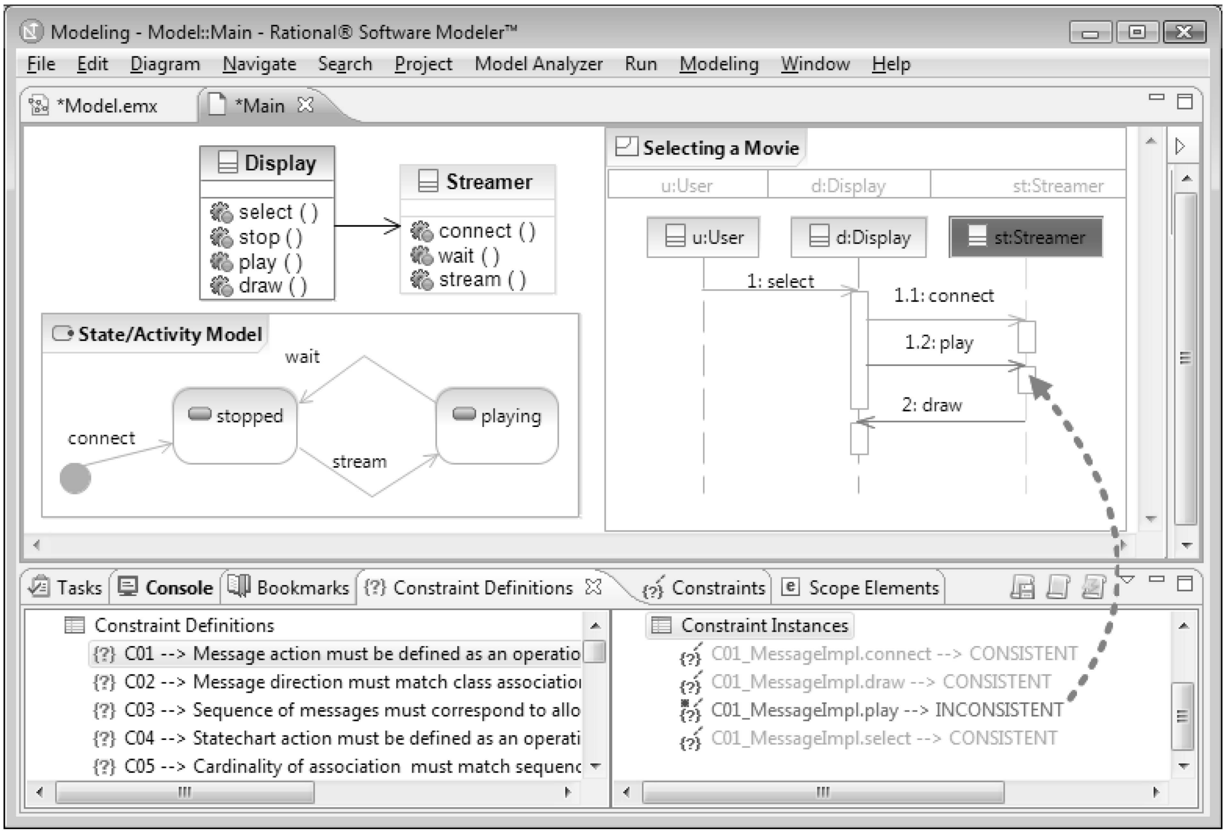
Fig. 8. Filtering CRIs in the rule detector to support faster instant and lazy consistency checking.

3.4 Tracking Inconsistencies

While it is important to know about inconsistencies, it is often too distracting to resolve them right away. The notion of "living with inconsistencies" [1], [5] advocates that there is a benefit in allowing inconsistencies in design models on a temporary basis. While our approach detects inconsistencies instantly, it does not require the engineers to fix them instantly. Our approach tracks all presently known inconsistencies and lets the engineers explore inconsistencies according to their interests in the model. If an engineer later on desires to identify all inconsistencies related to a particular model element (or set of model elements), then our approach simply searches through the scopes of all CRIs to identify the ones that are relevant. In [13], [15], we discuss how our technology helps the engineer resolve inconsistencies at some later time. This issue is out of the scope of this paper.

4 Model/Analyzer Tool

The Model/Analyzer tool implements our instant consistency checking infrastructure (Fig. 9). There are four implementations at this point: We built consistency checkers for IBM Rational Rose, Matlab/Stateflow, IBM Rational Software Modeler, and the Dopler product line tool suite [9]. The first two implementations are based on the UML13 infrastructure described in [14], the third implementation is based on the EMF (Eclipse Modeling Framework), and the fourth implementation is a nonstandardized modeling language. The diverse nature of these implementations is a strong support to our claim that our approach is independent of the modeling language. Our approach is also independent of the consistency rule language because we have implemented consistency rules in Java, J#, C#, and OCL [38] at this point. Fig. 9 depicts the screen snapshot for the IBM Rational Software Modeler implementation. The top depicts the illustration. Several inconsistencies are highlighted in red and the scope elements of one of the inconsistencies (consistency rule 1 evaluated on message *play*) are depicted below. The consistency rules are listed in the bottom left. The CRIs for the selected, first consistency rule are depicted in the bottom right. As was discussed earlier, the tool also helps the engineer in understanding exactly how model elements affect inconsistencies. As such, when the engineer selects a model element, say, the message *connect*, then the tool presents all CRIs that accessed it. This bidirectional navigation is essential for understanding and resolving inconsistencies.



Constraint Definitions
C01 --> Message action must be defined as an operatio
C02 --> Message direction must match class associatio
C03 --> Sequence of messages must correspond to allo
C04 --> Statechart action must be defined as an operati
C05 --> Cardinality of association must match sequenc

Constraint Instances
C01_MessageImpl.connect --> CONSISTENT
C01_MessageImpl.draw --> CONSISTENT
C01_MessageImpl.play --> INCONSISTENT
C01_MessageImpl.select --> CONSISTENT

Fig. 9. Model/analyzer tool depicting an inconsistency in IBM Rational Software Modeler.

This tool essentially automates all of the difficulties of instant consistency checking discussed in this paper and it was used for the empirical evaluation discussed in Section 5. To include new consistency rules, the tool provides an extension point for adding/removing consistency rules at any point in the design process.

In order to port our technology to other modeling languages, three basic challenges have to be addressed:

- Change notifications: In order to react to model changes, changes must be observable. Fortunately, most modeling tools today are open enough to allow this. In the cases of IBM Rational Rose and Matlab/Stateflow (older but still widely used tools), we were able to use probing mechanisms to elicit them. This is discussed in [14].
- Model profiling: In order to observe consistency checkers, access to model elements must be observable. We found that most modeling tools today *do not* provide adequate mechanisms to allow this. However, with change notification in place, we always managed to create an observable proxy of a tool's model such that consistency checkers accessed the tool's nonobservable model via the observable proxy. The proxy was easily maintainable through change notifications and typically required 1-3 person months to implement.
- Consistency rules: Different modeling languages may require different consistency rules. Fortunately, our approach does not prescribe special handling of consistency rules, aside from saying that if the consistency rules are written monolithically, then they should be divided up. We never encountered serious problems with this task.

With change notifications, model profiling, and the consistency rules in place, the remaining implementation of our incremental consistency checker was quite easily portable. The four implementations we have created to date (IBM Rational Rose, IBM Rational Software Modeler, Matlat/Stateflow, and DOPLER Product Lines) differ only in the parts that address the above three challenges and in certain basic assumptions on how to navigate a model or deal with certain data types (e.g., collections).

5 Validation

Instant consistency checking is only feasible if its computational cost is small and its results are correct. We thus empirically validated our approach on 34 UML models ranging from small models to very large ones (Table 1). These models were evaluated on 24 types of consistency rules (Fig. 2). In total, over 290,826 CRIs were evaluated, which accessed a total of 280,801 unique model elements or 830,603 model element-field pairs.

Fig. 3 previously presented the average response times of our approach relative to the model size. It showed that brute-force consistency checking was not instant. It also showed that type-based consistency checking did not scale either. And it showed that our approach was not noticeably affected by the model size for the 24 consistency rules.

TABLE 1
Study Models Used for Empirical Evaluation

Model Name	Class	Sequence	Statechart	# Model Elements	# Scope Elements
Video on Demand (Paper)	X	X	X	43	183
Microwave Oven	X	X	X	110	486
ATM	X	X	X	145	633
<unnamed1>	X	X		219	760
eBullition	X	X		243	960
Course Registration	X		X	286	993
Curriculum Planner	X	X		289	1154
Dice Game	X	X	X	387	1444
Inventory and Sales	X	X		415	1551
Teleoperated Robot	X	X	X	463	1585
NZ International Airport	X			495	1711
Home Appliances Control	X	X	X	441	1730
HDCP Defect Seeding	X	X	X	525	1863
<unnamed2>	X	X	X	512	1931
Vacation and Sick Leave	X	X	X	633	2449
ANTS Visualizer	X	X	X	881	2639
<unnamed3>	X	X	X	697	2869
NPI	X	X	X	922	3141
Building Management	X	X	X	862	3299
iTalks	X	X	X	984	3521
Video on Demand	X	X	X	1342	3780
DESI 2.3	X			1346	3930
Hotel Management	X	X	X	1025	4044
Biter Robocup Client	X			1916	5571
Calendarium 2.1	X	X	X	1538	5782
<unnamed4>	X	X	X	2171	6911
dSpace 3.2	X			5478	18075
Word Pad	X			6807	18755
Insurance Fees & Claims	X		X	10146	25064
OODT 07	X			9961	36128
<unnamed5>	X	X	X	16062	57703
<unnamed6>	X	X	X	18617	65045
<unnamed7>	X	X	X	32603	108981
<unnamed8>	X	X	X	162237	435932

These data were computed by systematically changing all model elements of all models. Since there were over 830,603 field values affected (most model elements had multiple fields), we did so automatically. Fig. 10 depicts the evaluation time across this large set of changes. We see that the evaluation time was less than 7 ms for 95 percent of all model changes—with a median of 0.2 ms. We consider an evaluation time of 100 ms or more to be noticeable by humans. In total, there were only 93 changes (out of the 830,603 total changes) where the evaluation time was above 100ms—with a worst case of 2 seconds. The measurements were made on an Intel processor with 2.2 GHz.

Our approach requires the existence of a scope for every CRI. The cost of computing the scope is negligible as it is already included in the evaluation times mentioned above.

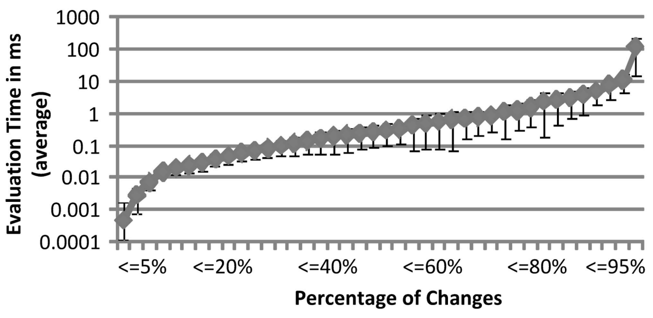


Fig. 10. Evaluation time of model changes across all CRIs.

However, for continuous use across modeling sessions, we either need to store the scope persistently (i.e., in a database) or recompute it every time a model is loaded. Both options are viable. The recomputation option is a one-time expense upon loading. The persistent storage option is also a one-time expense and reasonable as the scope database only grows linearly with the size of the model. We will first discuss the computation cost of loading and incremental reevaluation. Later, we discuss the memory cost of storing and maintaining CRIs.

- **Initial Cost of Computing All Truth Values and Scopes:** The cost of a single evaluation of a CRI is approximately the number of fields visited (= scope size S_{size}). The number of CRIs of a rule type $RT\#$ is at most the number of existing model elements M_{size}. The computational complexity for evaluating all CRIs is thus O($RT\#\ {}^{*}M_{size}{}^{*}S_{size}$). This cost is a one-time expense upon model load (if the scope is not maintained persistently).
- **Recurring Cost of Computing Changed Truth Values and Scopes:** For every changed model element, it is necessary to identify all CRIs that are affected. We define the number of affected CRIs as *ACRI*. The computational cost for evaluating all affected CRIs is thus O($ACRI^{*}S_{size}$). This cost is a recurring cost because it applies to every model change.

5.1 Computational Scalability

We applied our instant consistency checking tool (the Model/Analyzer) to the 34 sample models and measured the scope sizes S_{size} and the *ACRI* by considering all possible model changes. This was done through automated validation by systematically changing all fields of all model elements. In the following, we present empirical evidence that S_{size} and *ACRI* are small values that do not increase with the size of the model.

We expected some variability in S_{size} because the sample models were very diverse in contents, domain, and size. Indeed, we measured a wide range of values between the smallest and largest S_{size} (average/max), but found that the averages stayed constant with the size of the model. Fig. 11 depicts the values for S_{size} relative to the model sizes for the 34 sample models. The figure depicts each model as a vertical range (average to 98 percent maximum), where the solid dots are the average values for any given model. Notice the constant, horizontal line of average scope sizes.

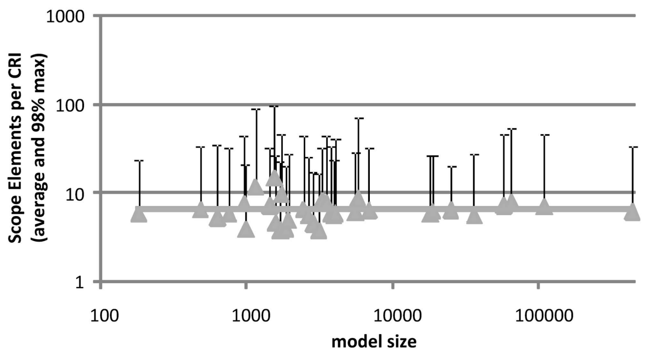


Fig. 11. CRI scope sizes remain constant with model sizes.

The initial, one-time cost of computing the truth values and scopes of a model is thus linear with the size of the model and the number of rule types $O(\mathrm{RT}^{*} M_{size})$ because S_{size} is a small constant and constants are ignored for computational complexity.

To validate the recurring computational cost of computing changed truth values and scopes, we next discuss how many CRIs must be evaluated with a single change (*ACRI*). Since the scope sizes were constant, it was expected that the *ACRI* would be constant also (i.e., the likelihood for CRIs to be affected by a change is directly proportional to the scope size). Again, we found a wide range of values for *ACRI* across the many diverse models but confirmed that the averages stayed constant with the size of the model. Fig. 12 depicts the average *ACRI* through solid dots and their 98 percent maximums.

ACRI was computed by evaluating all CRIs and then measuring in how many scopes each model element appeared. The figure shows that in some cases, many CRIs had to be evaluated (hundreds and more). But the average values reveal that most changes required few evaluations (between 3 and 11 depending on the model).

Fig. 13 depicts the average cost of evaluating a model change based on the type of change (Fig. 13a). We see that a change to the *association* field of an *AssociationEnd* was the most expensive kind of change, with over 4 ms reevaluation cost, on average. A message name change (as was used several times in this paper) was comparatively cheap, with 0.12 ms to reevaluate, on average. First and foremost, we note that all types of model changes are quite reasonable to reevaluate. This implies that irrespective of how often certain types of changes happen, our approach performs

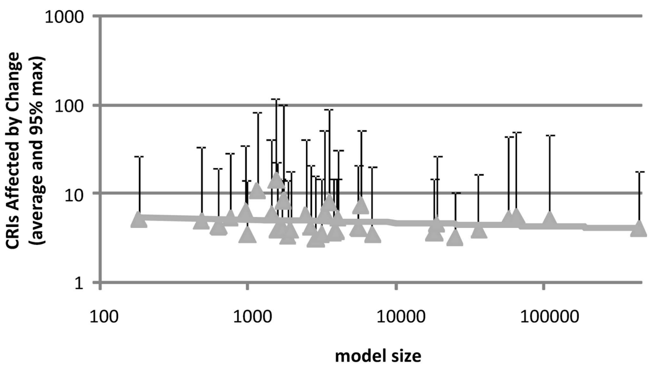


Fig. 12. Few consistency rule instances are affected by a model change.

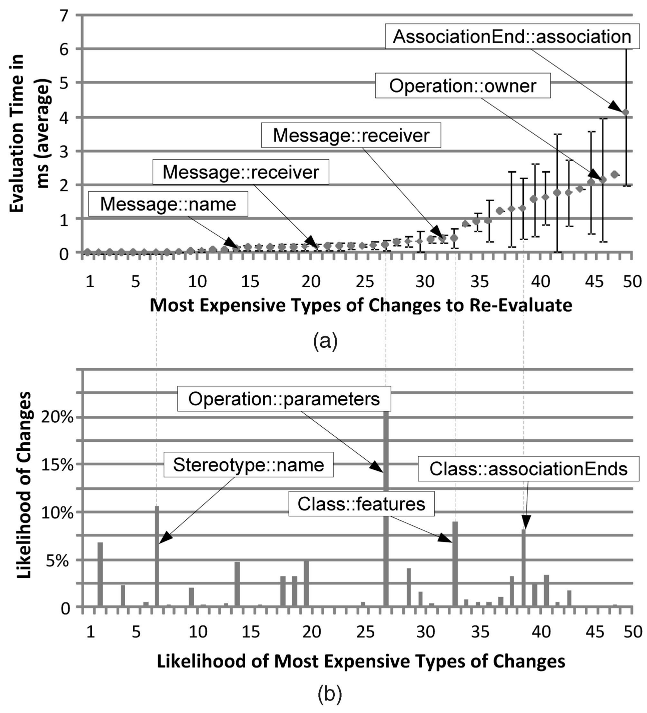
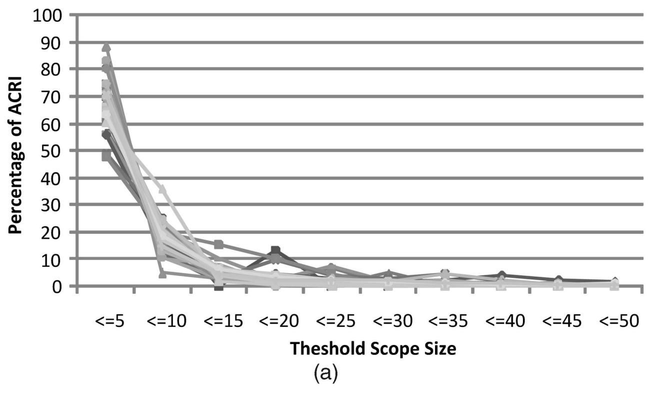


Fig. 13. The most expensive types of model changes to evaluate and the likelihoods of these changes occurring.

well on all of them. However, not all changes are equally likely and we thus investigated the likelihood of these most expensive types of model changes. For 8 out of the 34 models, we had access to multiple model versions—covering 4,075 changes across them. Fig. 13b depicts that the model changes were unevenly distributed across the types, but as was expected, there is no single (or few) dominant kinds of model changes. Indeed, the most expensive types of model changes never occurred.

Previously, we mentioned that most changes required very little reevaluation time and that there were very rare outliers (0.00011 percent of changes with evaluation time >100 ms). The reason for this is obvious in Fig. 14, where we see that it is exponentially unlikely for CRIs to have larger scope sizes (Fig. 14a) or for changes to affect many CRIs (Fig. 14b). We show this datum to exemplify how similar the 34 models are in that regard, even though these models are vastly different in size, complexity, and domain. Fig. 14a depicts for all 34 models separately what percentage of CRIs (y-axis) had a scope of $<= 5, 10, 15, \ldots$ scope elements (x-axis). The table shows that over 95 percent of all CRIs accessed less than 15 fields of model elements (scope elements). Fig. 14b depicts for all 34 models separately what percentage of changes (y-axis) affected $<= 2, 4, 6, \ldots$ CRIs. The table shows that 95 percent of all changes affected fewer than 10 CRIs (*ACRI*).

The data thus far considered a constant number of consistency rules (24 consistency rules). However, the number of consistency rules is variable and may change from model to model or domain to domain. Clearly, our approach (or any approach to incremental consistency checking) is not amendable to arbitrary consistency rules. If a rule must investigate all model elements, then such a rule's scope is bound to increase with the size of the model. However, we demonstrated on the 24 consistency rules that



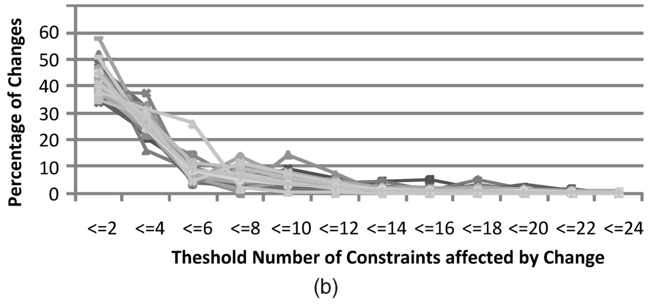


Fig. 14. (a) The number of model elements accessed by constraints and (b) the number of constraints affected by changes as percentages relative to thresholds.

rules typically are not global; they are, in fact, surprisingly local in their investigations. This is demonstrated in Fig. 15, which depicts the cost of evaluating changes for each consistency rule separately. Still, each consistency rule takes time to evaluate and Fig. 15 is thus an indication of the increase in evaluation cost in response to adding new consistency rules. We see that the 24 consistency rules took, on average, 0.004-0.21 ms to evaluate with model changes. Each new consistency rule thus increases the evaluation time of a change by this time (assuming that new consistency rules are similar to the 24 kinds of rules we evaluated). The evaluation time thus increases linearly with the number of consistency rules (*RT#*).

It is important to note that the evaluation was based on consistency rules implemented in C#. Rules implemented in Java were slightly slower to evaluate but rules implemented in OCL [38] were comparatively expensive due to the high cost of interpreting them.

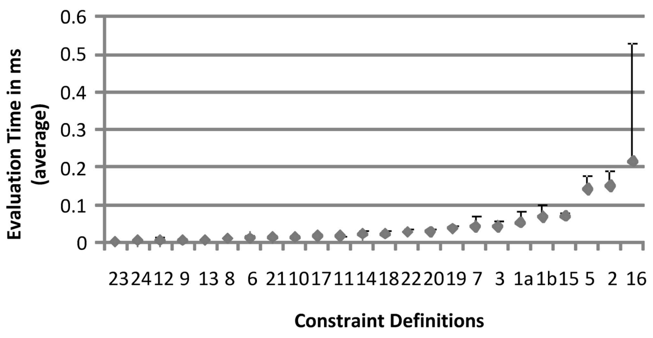


Fig. 15. The cost of adding a consistency rule.

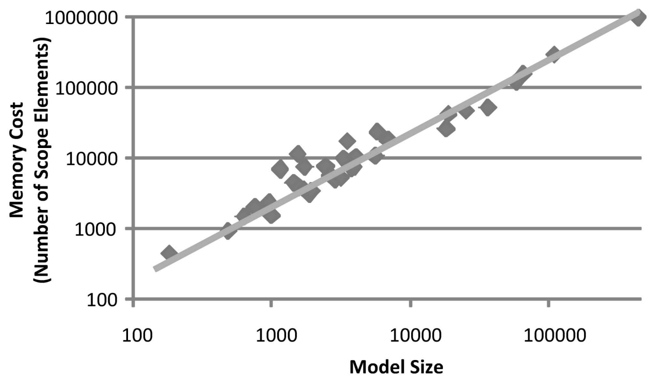


Fig. 16. Memory cost increases linearly with model size.

5.2 Memory Cost

On the downside, our approach does require additional memory for storing the scopes. Fig. 16 depicts the linear relationship between the model size and this memory cost. It can be seen that the memory cost rises linearly. This should not be surprising given that the scope sizes are constant with respect to the model size but the number of CRIs increases linearly. As with the evaluation time, this cost also increases with the number of consistency rules (*RT#*). The memory cost is thus RT# $^*S_{size}$. For scalability, this implies a quite reasonable trade-off between the extensive performance gains over a linear (and thus scalable) memory cost. To put this rather abstract finding into a practical perspective, the scope is maintained as a simple hashtable referencing the impacted CRIs in form of arrays. With the largest model having over 400,000 scope elements, each of which affects fewer than 10 CRIs, the memory cost is thus equivalent to 400,000 arrays of fewer than 10 CRIs each—quite manageable with today's computing resources. The memory cost stays the same if the scope is stored persistently, in which case the recomputation of the scope upon model load is no longer required.

5.3 Usability

One key advantage of our approach is that engineers are not limited by the modeling language or consistency rule language. We demonstrated this by implementing our approach on UML 1.3, UML 2.1, Matlab/Stateflow, and Dopler Product Line, and using a wide range of languages to describe consistency rules (from Java, C# to the interpreted OCL). But, most significantly, engineers do not have to understand our approach or provide any form of manual annotations (in addition to writing the consistency rule) to use it. These freedoms are all important for usability.

This paper does not address how to best visualize inconsistencies graphically. Much of this problem has to do with human-computer interaction and future work will study this. This paper also does not address downstream economic benefits: For example, how does quicker (instant) detection of inconsistencies really benefit software engineering at large. How many problems are avoided, how much less does it cost to fix an error early on as compared to later on? These complex issues have yet to be investigated. However, as an anecdotal reference, it is worth pointing out that nearly all programming environments today support instant compilation (and thus syntax and semantic checking), which clearly benefits programmers. We see no reason why these benefits would not apply to modeling.

5.4 Threats to Validity

Internal validity. We investigated 24 consistency rules in the context of 34, mostly third-party, UML models. Both models and rules were very diverse in size and domain. The consistency rules we used were based on the standard literature. We did not discard any rules or models as outliers and we evaluated the impact of changes across all of them exhaustively. Since our approach performed well for all of these models and rules, we believe that the threats to internal validity are small.

External validity. While the evaluation focused on the UML 1.3 notation due to the availability of large design models, we have tested our infrastructure on UML 2.1, Matlab/Stateflow, and the Dopler Product Line. While the models available there were not as large as the ones used in this study, we were able to confirm that the approach works and also scales—including the Dopler language, which is semantically very different from UML and had very different kinds of consistency rules.

However, more consistency rules imply more evaluation time. This cost is expected to increase linearly. Clearly, we cannot support an infinite number of constraint rules but we typically do not have to. For the engineers we worked with, the 24 rules covered all of relevant situations for the consistency of sequence diagrams with class and statechart diagrams (in their domains). And there are a few hundred other known rules for other types of UML diagrams, say, deployment diagrams or use-case diagrams. Thus, even if these other rules were included in our approach, the scopes of these rules would mostly overlap with other UML diagrams and thus not affect our rules much. This implies that more consistency rules do not necessarily imply longer evaluation times. However, given that all consistency rules evaluated added less than 1 ms each to the total evaluation time, we do not foresee scalability issues even with one or two orders of magnitude larger RT#.

This is not to say that consistency rules are always as local as discovered here. We believe that there could be rules that would not scale well. However, given that we looked at a fairly large number of rules for different languages, we are confident that most rules would scale. An example of rules that would be more complex to validate is the rules that require extensive transformation in addition to checking. For example, validating the correct refinement of two class diagrams is an issue we explored in [35]. This problem requires not only consistency checking (in the traditional sense) but also model transformation (to abstract low-level models to high-level models [11]). Yet, even in such situations, it is possible to increase efficiency significantly by separating transformation from comparison (consistency checking).

6 Related Work

While researchers generally agree on what consistency means, the methods on how to detect (in)consistencies vary widely. In essence, we see a division between those who compare design models directly and those who transform design models into some intermediate representation to compare there.

For example, Tsiolakis and Ehrig [19] check the consistency between class and sequence diagrams by converting both into a common graph structure. VisualSpecs [3] uses transformation to substitute the imprecision of OMT (a language similar to UML) with algebraic specifications. Conflicting specifications are then interpreted as inconsistencies. Belkhouche and Lemus [2] also follow along the tracks of VisualSpecs in their use of a formal language to substitute statechart and dataflow diagrams. Groher et al. [20] explore the use of description logic to detect inconsistencies between sequence and statechart diagrams. Campbell et al. [7] make use of the SPIN Model checker to evaluate a range of consistency problems within and across UML diagrams. Or, Zisman and Kozlenkov [40] use a knowledge base where, with the help of patterns and axioms, consistency rules are expressed. Using an intermediate representation has many advantages. Yet, for instant consistency checking, it has the disadvantage of also having to implement incremental transformation in addition to incremental consistency checking to continuously translate and compare model changes. Furthermore, it has the problem that inconsistencies detected in the intermediate representation have to be transformed back to make it understandable to the engineer—who ideally should not even be aware of the intermediate representation. To the best of our knowledge, to date there exists only one approach to incremental consistency checking that is based on intermediate models. However, this approach separated the transformation and comparison for the sake of speeding up consistency checking [35] in situations where transformation is computationally expensive and/or its changes cannot easily be synced with that of comparison. This approach is quite complex and requires manual overhead in defining consistency and transformation rules—an overhead that is avoidable for many consistency rules, as was shown in this paper.

The use of an intermediate representation is not a prerequisite for consistency checking. Indeed, it is possible to write a consistency rule that directly compares design models rather than transforming them first [18], [21], [27], [31]. The most interesting ones among these are the xLinkIt [27] and ArgoUML [31] approaches because they are capable of performing incremental consistency checking.

xLinkIt [27] is an XML-based environment for evaluating the consistency of "documents." Such documents could be anything, including UML design models. The advantage of this environment is that consistency rules are expressed in a uniform manner. xLinkIt is capable of checking the consistency of an entire UML model and it also handles incremental consistency by only evaluating changes to versions of a "document." However, it requires between 5 and 24 seconds for evaluating changes and thus is not able to keep up with an engineer's rate of model changes. It is most useful for the occasional exchange of models and for enforcing consistency constraints in a uniform manner across different modeling languages. The approach by Reiss [30] is, in principle, alike xLinkIt. Rather than defining consistency rules on XML documents, Reiss defines consistency rules as SQL queries, which are then evaluated on a database which may hold a diverse set of artifacts. Reiss' use of a database certainly makes his approach more incremental. However, the incremental updates in his study suggest noninstant performance (with 30 seconds to 3-minute build times). However, his models are more distributed and also include queries on code, and thus his work appears applicable in maintaining artifacts for a diverse tool set, which our approach or xLinkIt does not.

ArgoUML also detects inconsistencies in UML models [31] but it requires annotated consistency rules to enable incremental consistency checking. ArgoUML implements two consistency checking mechanisms: a "warm queue" and a "hot queue." Consistency rules for which no annotations are provided are placed into the warm queue. This queue is continuously evaluated at 20 percent CPU time. Consistency rules in the hot queue have annotations as to what types of model elements they affect. If a model element changes, then all of those consistency rules are evaluated that are affected by that element's type. We demonstrated that type-based consistency checking produces good performance but it is not able to keep up with an engineer's rate of model changes in very large models. Also, it requires additional annotations, which are not required by our approach. The evaluation of the warm queue is essentially batch consistency checking, and thus not scalable for even moderately large models. Yet ArgoUML is an excellent tool for visually presenting instant consistency feedback in a nonintrusive manner. This aspect of ArgoUML is directly applicable to our approach. It is important to note that ArgoUML's way of treating consistency checking has been adopted in a range of commercial modeling tools: For example, the IBM Rational Software Modeler also uses a type-based scope to incremental consistency checking, which is significantly better than batch consistency checking but still far from as instant as our approach.

Blanc et al. [4] approach the issue of incremental consistency checking from the perspective of model changes. This is, in principle, like our approach since we also react to changes. However, their consistency rules are defined explicitly in terms of their impact on changes. This requires the engineer to annotate consistency rules with the exact impact of all design changes. This annotation, if done correctly, leads to good performance. However, since writing these annotations may cause errors, they are no longer able to guarantee the correctness of incremental consistency checking. Our approach, on the other hand, does not impose any manual overhead on the engineer. This source of error is thus eliminated.

While it is important to know about inconsistencies, it is often too distracting to resolve them right away. The notion of "living with inconsistencies" [2], [17] advocates that there is a benefit in allowing inconsistencies in design models on a temporary basis. While our approach provides inconsistencies instantly, it does not require the engineer to fix them instantly. Our approach tracks all presently known inconsistencies and lets the engineer explore inconsistencies according to his/her interests in the model. This is a nontrivial problem because the scope of an (in)consistency is continuously affected by model changes. Our approach could also be used for lazy consistency checking, which has been explored in [32] but is out of the scope of this paper. Also out of the scope of this paper is the issue of how to fix inconsistencies. While inconsistencies may be tolerated for some time, fixing them is eventually necessary. This issue is explored in [13], [15], [28], [39].

Viewpoints [10] is a classical approach to consistency checking. It also uses consistency rules that are defined and validated against a formal model base. This approach, however, emphasizes "upstream" modeling techniques and it addresses issues such as how to resolve inconsistencies [13], [28], [29], [34] and how to tolerate them. These aspects are not discussed in this paper but are very relevant to consistency checking. It is future work to discuss how our approach handles these aspects.

In a broader sense, current approaches to consistency checking borrow from programming environments such as Centaur or Gandalf [22] that incrementally evaluate syntactic or even semantic [16] consistency rules within source code. These approaches use grammar information to generate programming environments and incremental consistency checker. In the UML domain, consistency rules and checkers often already exist and are not generated. While engineers in industry (e.g., Boeing Company) do create their own consistency rules, they usually do not use a grammar-based language. Yet they also investigate decentralized consistency checking [23] and consistency checking among different languages, which is considered outside the scope here [36].

Our work is loosely related to the constraint satisfaction problem (CSP). CSP deals with the combinatorial problem of what choices best satisfy a given set of constraints. Since this problem is computationally expensive, certain optimizations have been developed. In particular, the AC3 optimization [26] defines a mapping between choices and the constraints they affect. Constraints are then reevaluated only if their choices change. We borrowed this concept in our use of scopes. A key difference is that CSP uses "white box constraints" where it is known in advance what choices a constraint will encounter. This makes it relatively easy to identify their scopes. Consistency rules in UML typically are black box constraints. This is the main reason why most approaches to incremental consistency checking require additional annotations for consistency rules to cope with this additional level of indirection.

Furthermore, it is worthwhile to stress that incremental reasoning is used in many domains outside of consistency checking. A considerable community exists that investigates the effect of changes. Indeed, we can see some parallels between our approach and the classical RETE algorithm [19], which uses pattern matching as a means of establishing correspondence between facts (e.g., models) and patterns (e.g., consistency rules). Another example is the self-adaptive computation [1], which applies to source code and investigates how to update algorithms to better react to data changes. Since our consistency rules can be seen as such algorithms, a similarity exists. However, it is important to point out that to the best of our knowledge, nobody has tried to apply/transform these principles to the consistency checking of software modeling. Also, while the concepts have similarities, the details of these approaches do differ widely from our approach.

7 Conclusion

This paper introduced an approach for quickly, correctly, and automatically deciding when to evaluate consistency rules. We demonstrated that our approach works with many consistency rules and that these rules do not have to be written in a special language with special annotations. Instead, our approach used a form of profiling to observe the behavior of the consistency rules during evaluation. We demonstrated on 34 large-scale models that the average model change cost 1.4 ms, 98 percent of the model changes cost less than 7 ms, and that the worst case was below 2 seconds.

It is very significant to understand that our approach maintains a separate scope of model elements for every application (instance) of a consistency rule. This scope is computed automatically during evaluation and used to determine when to reevaluate the rule. In the case of an inconsistency, this scope tells the engineer all of the model elements that were involved. Moreover, if an engineer should choose to ignore an inconsistency (i.e., not resolve it right away), an engineer may use the scopes to quickly locate all inconsistencies that directly relate to any part of the model of interest. This is important for living with inconsistencies but it is also important for not getting overwhelmed with too much feedback at once.

However, we cannot guarantee that all consistency rules can be evaluated instantly. The 24 rules of our study were chosen to cover the needs of our industrial partners. They cover a significant set of rules and we demonstrated that they were handled extremely efficiently. But it is theoretically possible to write consistency rules in a nonscalable fashion, although it must be stressed that of the hundreds of rules known to us, none fall into this category.

It is future work to discuss how to best present inconsistency feedback visually to the engineer. Also, the efficiency of our approach depends, in part, on how consistency rules are written. Since consistency rules are typically written manually (by engineers), it is future work to investigate how to automatically optimize consistency rules. We believe that it is possible to automatically transform more complex, global consistency rules into more numerous and efficient local consistency rules.

Acknowledgments

The author would like to thank Iris Groher and Alexander Reder for porting an earlier version of the tool onto IBM Rational Software Modeler. This work was supported by the Austrian FWF under agreement P21321-N15.

References

[1] U.A. Acar, A. Ahmed, and M. Blume, "Imperative Self-Adjusting Computation," *Proc. 35th ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages*, pp. 309-322, 2008.

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

[3] B. Belkhouche and C. Lemus, "Multiple View Analysis and Design," *Proc. Int'l Workshop Multiple Perspectives in Software Development*, 1996.

[4] X. Blanc, I. Mounier, A. Mougenot, and T. Mens, "Detecting Model Inconsistency through Operation-Based Model Construction," *Proc. 30th Int'l Conf. Software Eng.*, pp. 511-520, 2008.

[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] L.C. Briand, Y. Labiche, and L. O'Sullivan, "Impact Analysis and Change Management of UML Models," *Proc. Int'l Conf. Software Maintenance*, p. 256, 2003.

[7] L.A. Campbell, B.H.C. Cheng, W.E. McUmber, and K. Stirewalt, "Automatically Detecting and Visualising Errors in UML Diagrams," *Requirements Eng. J.*, vol. 7, pp. 264-287, 2002.

[8] B.H.C. Cheng, E.Y. Wang, and R.H. Bourdeau, "A Graphical Environment for Formally Developing Object-Oriented Software," *Proc. Sixth Int'l Conf. Tools with Artificial Intelligence*, pp. 26-32, 1994.

[9] D. Dhungana, R. Rabiser, P. Grünbacher, K. Lehner, and C. Federspiel, "DOPLER: An Adaptable Tool Suite for Product Line Engineering," *Proc. 11th Int'l Software Product Line Conf.*, pp. 151-152, 2007.

[10] S. Easterbrook and B. Nuseibeh, "Using ViewPoints for Inconsistency Management," *IEE Software Eng. J.*, vol. 11, pp. 31-43, 1995.

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

[12] A. Egyed, "Instant Consistency Checking for the UML," *Proc. 28th Int'l Conf. Software Eng.*, pp. 381-390, 2006.

[13] A. Egyed, "Fixing Inconsistencies in UML Design Models," *Proc. 29th Int'l Conf. Software Eng.*, pp. 292-301, 2007.

[14] A. Egyed and B. Balzer, "Integrating COTS Software into Systems through Instrumentation and Reasoning," *Int'l J. Automated Software Eng.*, vol. 13, pp. 41-64, 2006.

[15] A. Egyed, E. Letier, and A. Finkelstein, "Generating and Evaluating Choices for Fixing Inconsistencies in UML Design Models," *Proc. 23rd Int'l Conf. Automated Software Eng.*, 2008.

[16] W. Emmerich, "GTSL—an Object-Oriented Language for Specification of Syntax Directed Tools," *Proc. Eighth Int'l Workshop Software Specification and Design*, pp. 26-35, 1996.

[17] S. Fickas, M. Feather, and J. Kramer, *Proc. ICSE-97 Workshop Living with Inconsistency*, 1997.

[18] 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.

[19] C. Forgy, "Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem," *Artificial Intelligence*, vol. 19, pp. 17-37, 1982.

[20] I. Groher, A. Reder, and A. Egyed, "Instant Consistency Checking of Dynamic Constraints," *Proc. 12th Int'l Conf. Fundamental Approaches to Software Eng.*, 2010.

[21] 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.

[22] A.N. Habermann and D. Notkin, "Gandalf: Software Development Environments," *IEEE Trans. Software Eng.*, vol. 12, no. 12, pp. 1117-1127, Dec. 1986.

[23] S.M. Kaplan and G.E. Kaiser, "Incremental Attribute Evaluation in Distributed Language-Based Environments," *Proc. Fifth Ann. Symp. Principles of Distributed Computing*, pp. 121-130, 1986.

[24] M. Lee, A.J. Offutt, and R.T. Alexander, "Algorithmic Analysis of the Impacts of Changes to Object-Oriented Software," *Proc. 34th Int'l Conf. Technology of Object-Oriented Languages and Systems*, pp. 61-70, 2000.

[25] M. Lindvall and K. Sandahl, "Practical Implications of Traceability," *J. Software—Practice and Experience*, vol. 26, pp. 1161-1180, 1996.

[26] A.K. Mackworth, "Consistency in Networks of Relations," *J. Artificial Intelligence*, vol. 8, pp. 99-118, 1977.

[27] 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.

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

[29] B. Nuseibeh and A. Russo, "On the Consequences of Acting in the Presence of Inconsistency," *Proc. Ninth Int'l Workshop Software Specification and Design*, pp. 156-158, 1998.

[30] S. Reiss, "Incremental Maintenance of Software Artifacts," *IEEE Trans. Software Eng.*, vol. 32, no. 9, pp. 682-697, Sept. 2006.

[31] J. Robins et al, "ArgoUML," http://argouml.tigris.org, 2010.

[32] N. Roussopoulos, "An Incremental Access Method for View-Cache: Concept, Algorithms, and Cost Analysis," *ACM Trans. Database Systems*, vol. 16, pp. 535-563, 1991.

[33] J. Rumbaugh, J. Ivar, and B. Grady, *The Unified Modeling Language Reference Manual.* Addison Wesley, 1999.

[34] M. Sabetzadeh, S. Nejati, S. Liaskos, S. Easterbrook, and M. Chechik, "Consistency Checking of Conceptual Models via Model Merging," *Proc. 15th IEEE Int'l Requirements Eng. Conf.*, 2007.

[35] W. Shen, K. Wang, and A. Egyed, "An Efficient and Scalable Approach to Correct Class Model Refinement," *IEEE Trans. Software Eng.*, vol. 35, no. 4, pp. 515-533, July/Aug. 2009.

[36] R.N. Taylor, R.W. Selby, M. Young, F.C. Belz, L.A. Clarce, J.C. Wileden, L. Osterweil, and A.L. Wolf, "Foundations of the Arcadia Environment Architecture," *Proc. Fourth Symp. Software Development Environments*, 1998.

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

[38] J. Warmer and A. Kleppe, *The Object Constraint Language.* Pearson Education, 2003.

[39] Y. Xiong, Z. Hu, H. Zhao, H. Song, M. Takeichi, and H. Mei, "Supporting Automatic Model Inconsistency Fixing," *Proc. Seventh Joint Meeting of the European Software Eng. Conf. and the ACM SIGSOFT Symp. Foundations of Software Eng.*, 2009.

[40] A. Zisman and A. Kozlenkov, "Knowledge Base Approach to Consistency Management of UML Specification," *Proc. 16th IEEE Int'l Conf. Automated Software Eng.*, pp. 359-363, 2001.

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 full 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, variability modeling, consistency checking/resolution, and change impact analysis. He is a member of the IEEE, the IEEE Computer Society, the ACM, and ACM SigSoft.