Automatically Detecting and Tracking Inconsistencies in Software Design Models

Alexander Egyed, *Member, IEEE*

***Abstract*—Consistency checkers help engineers find errors (inconsistencies) in software design models. 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 design models in real time (while the model changes). Engineers only need to define consistency rules – in any language and without any manual annotations as required by the current state-of-the-art. Our approach automatically identifies how model changes affect these consistency rules. It does this through model profiling during consistency checking to observe the behavior of consistency rules to understand how they affect the model (and are thus affected by model changes). The approach is quick, correct, scalable, fully automated, and also easy to use as it does not require any special skills from the engineers who want to use it. We evaluated the approach on 34 models with model sizes of up to 162,237 model elements and 24 types of consistency and well-formedness rules. Our empirical evaluation shows that our approach requires only 1.4 ms to re-evaluate the consistency of the model after a change (in average), its performance is not affected by the model size but only by the number of consistency rules, at the expense of a quite acceptable, linearly increasing memory consumption.**

***Index Terms*—D.2.10 [Design]**

I. 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 for the entire model [2, 7, 34]. Unfortunately, batch consistency checking does not scale and the checking of larger models can take hours to complete. As a result, designer use such batch consistency checkers occasionally only – waiting days perhaps

Manuscript received July 2009. This work was supported in part by the Austrian FWF under agreement P21321-N15
A. Egyed is with the Johannes Kepler University, Linz, Austria (telephone: +43 732 2468 8866, e-mail: alexander.egyed@jku.at).

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 [4]. Batch consistency checking thus cannot keep up with the engineers, provides late feedback, and interrupts the workflow of the engineers involved.

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 non-intrusive manner. Although, there are several modeling tools that support the incremental consistency checking of design models, they either do not provide design feedback instantly [24] or they require extensive manual overhead in annotating consistency rules [28]. This is because correctly deciding how model changes affect the consistency rules is a complex task given the very large number of possible changes involved. Any manual overhead in deciding this is also bound to be error prone and we will demonstrate that the kinds of filters used today to support incremental consistency checkers are not capable of providing design feedback in real-time either.

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 is used to establish a correlation between model elements and consistency rules (and inconsistencies). Based on this correlation, we decide when to re-evaluate 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 [1, 16]). 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 nor willing to define these additional declarations and/or annotations. This is particularly then a severe problem when engineers create their own variations of non-standard consistency rules, as is often done in UML [30]. **Engineers can thus define consistency rules at will, in any language, without any knowledge or training on our consistency checking mechanism (since it is completely hidden from the engineer).** Our approach is integrated into the commercial modeling tools IBM Rational Rose™ and 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, 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 shows that the evaluation of a model after changes averaged to less than 1.4 ms (in only 0.00011% of changes was the evaluation time >100 ms but never worse than 2 seconds) - even on the largest industrial models we had available. It is truly instant. 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 non-intrusive manner. It may also be coupled with inconsistency actions to resolve errors automatically [13, 14, 25]. However, space limitations preclude these discussions.

To date, our technology was applied to UML 1.3, UML 2.1, Matlab/Stateflow, and the Dopler product line to demonstrate that our approach is applicable to many modeling notations. We also implemented consistency checkers and rules using a range of different languages – Java, C#, J#, and OCL (Object Constraint Language [35]) – to demonstrate that our approach is applicable 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 [10].

II. PROBLEM

The following 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.

A. Consistency Rules and Illustration

The illustration in Figure 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

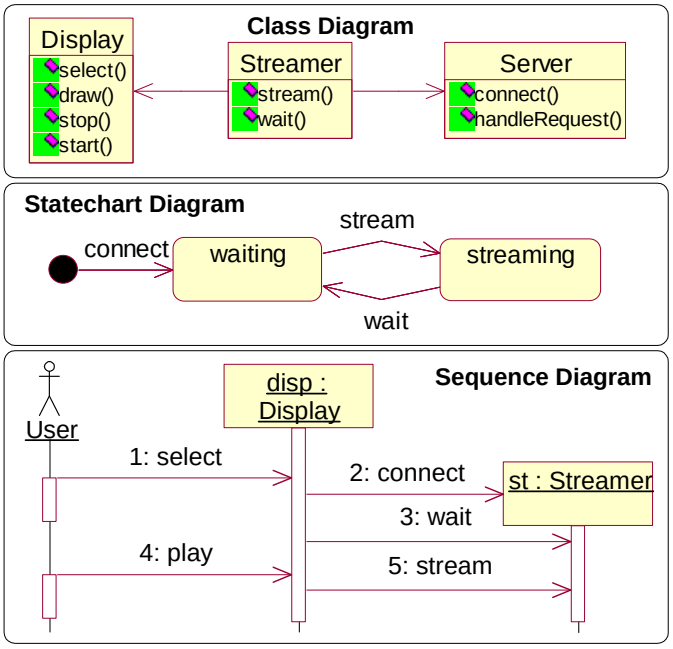

Figure 1. Simplified UML Model of the VOD System

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.

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*.

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(object sequence equal reachable sequence from startingPoint)
4	**Cardinality of association must match sequence interaction** for each instance for each associations of instance.base association.otherEnd.multiplicity > = count(instance.messages.receiver.base=association.otherEnd.type)
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 classifier 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**

Figure 2. Consistency Rules for UML Class, Sequence, and Statechart Diagrams. Details sketched for first four rules only. Rules 7 and 8 are classical best practice rules (and not necessarily errors). Rule 9-25 are typical UML well-formedness rules identified in the UML standards 1.3 and 2.1

These UML Consistency rules describe conditions that an UML model must satisfy for it to be considered a valid UML model. Figure 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. A consistency rule may be

thought of as a *condition* that evaluates a portion of a model to a *truth value*.

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 3rd 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*.

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). Or, the *disp* object calls the *st* object (arrow direction) 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 meta model element. For example, all messages (e.g., *wait*) are instances of the UML meta model element *Message*; and all objects (e.g., *st*) are instances of the UML meta model element *Object*.

$$\textbf{\textit{Consistency Rule}} =< Condition, Context\ Element >\rightarrow Boolean$$

$$\textbf{\textit{Context Element}} \in \textbf{\textit{Meta Model Elements}}$$

Consistency rules are typically evaluated from the point of view of a meta model 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. Consistency rules are certainly affected by changes to their context element, 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 consistency of any given consistency rule**.

B. Understanding Changes

Since consistency rules are conditions on a model, their truth values (the condition of a consistency rule is either true or false) change only if the model changes (or if the condition changes but this is typically not the case and not considered here). Instant consistency checking thus requires an 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 model element and it modifies the two existing objects:

- New model element of type UML.Message with ID 100
- Modified model element of type UML.Object [*outgoingMessages*] with ID 101
- Modified model element of type UML.Object [*incomingMessages*] with ID 102)

The first change notification tells about the creation of a model element – an instance of an 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.

$$\boldsymbol{Meta\ Model\ Element} = \bigcup_{\boldsymbol{n}} \boldsymbol{Field}$$

$$\boldsymbol{Field} = \{\boldsymbol{Basic\ Type}\ |\boldsymbol{Reference}\ (\boldsymbol{to\ Meta\ Model\ Element})\}$$

It is important to note that model elements are aggregations of fields. For example, an UML message has a *name* field (of type *string*) or a *receiver* field (a references to the UML *ClassifierRole*) to describe the properties of a message and its relationship to other model elements. In case of the UML, the meta model 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 a model element) and the validity of consistency rules is typically only affected by individual fields only**. For example, the creation of the message changed the *outgoingMessages* and *incomingMessages* fields of both objects only (not modified were, say, the *name* fields of both objects).

Changes identify the fields of model elements and it is thus obvious that the impact of a change onto consistency rules should also be described in terms of fields of model elements.

$$\boldsymbol{Change} =< ModelElement, Field >: \boldsymbol{oldValue} \rightarrow \boldsymbol{newValue}$$

Changes are described in terms of their impact on the model and not the action performed by the engineer. For example, an engineer may create a message in the modeling tool by invoking a menu item or by dragging a toolbar icon, yet the result, the creation, is the same. There is also no ordering in the changes caused by individual user actions. It is obvious that the new message much 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.

$$\boldsymbol{Change\ Group} = \bigcup_{\boldsymbol{n}} \boldsymbol{Change}$$

C. Understanding Impact of a Change

It is intuitive to think of instant consistency checking in terms of *what happens if* a model element changes [3]. 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 evaluation 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 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 in 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 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 thus rely on what we refer to as a type-based scope for incremental consistency checking. The problem is, however, that this scope must be pre-defined by the creator of the

consistency rule(s). Doing so is not a simple exercise since the engineer needs to annotate the consistency rule to identify what types of model elements affect the rule. This manually created type-based scope is used to automatically decide when to evaluate a consistency rule. If a model element changes then all those rules are evaluated that access this type of model element. For example, consistency rule 1, starts at a message (type *Message*), calls the receiver (type *Object*), then calls the base (type *Class*) and finally its operations (type *Operation*). The type-based scope for rule 1 is thus {Message, Object, Class, and Operation}.

While the type-based scope remains constant for any given consistency rule, it is obvious that the larger the model becomes the more instances of those types of model elements exist (e.g., a larger model will have a larger number of messages, objects, classes, and operations). For a larger model, consistency rule 1 thus needs to evaluate many model elements and, in reverse, a change to one of its model elements triggers many more consistency re-evaluations. Figure 3 depicts the results of empirically evaluating model changes on 34 sample models. It shows that batch consistency checking approach (one that evaluated all consistency rules) becomes expensive quickly (it is only instant with small-sized models of less than 1000 model elements). Approaches such as ArgoUML, which use type-based scopes, fare significantly better. Yet, we found that even such approaches do not scale well and are instant only with medium-sized models of up to 10,000 model elements (see Figure 3). While 10,000 elements may appear large, we must stress that the industrial models we worked with were typically larger – the largest having had 162,237 model element (and over 435,932 fields) for which the average consistency checking response time was seconds or more. It must also be stressed that these values are based on 24 consistency rules only. The computational cost increases linearly with the number of consistency rules involved and the UML alone defines hundreds of well-formedness rules. **The**

performance of type-based 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.

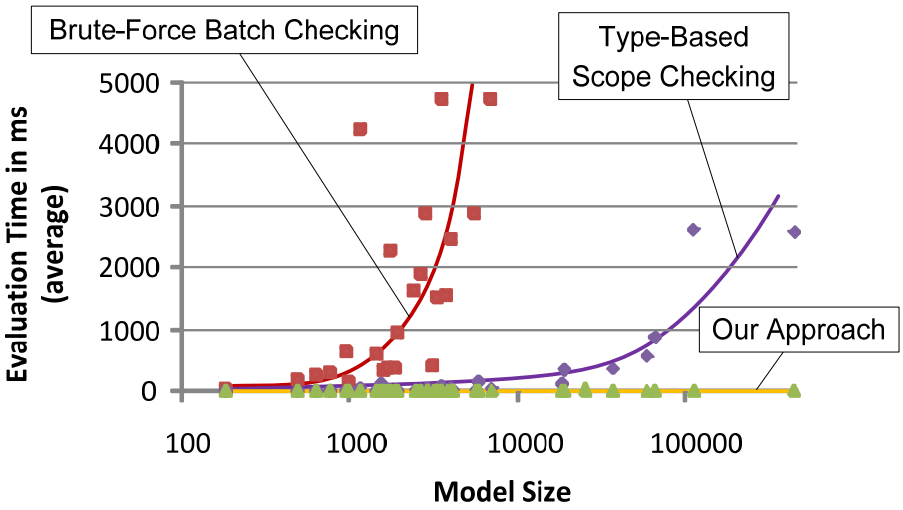

Figure 3. Evaluation Time per Model Change

We will show next that our approach is much faster, scalable (not affected by the size of the model), and does not require manual annotations for consistency rules (like the manually created type-based scope). Figure 3 depicts the performance of our instance-based approach on the same models. The average evaluation time per model change is in the range of milliseconds. But more significantly, the computational cost (performance) of our approach does not increase with the size of the model (it scales).

III. 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 meta model 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

re-evaluated if and only 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 short *scope*. The scope must be complete for our approach to be correct and the scope must be small for our approach to be quick. 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 was 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 run-time 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** (or short CRIs). Next, we discuss that the set of model elements accessed during a CRI's evaluation is in fact a superset of the minimal change impact scope.

A. 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 II.A). This behavior was defined in Figure 2. Rule 1 evaluated on message *wait* thus accesses the model elements {*wait*, *st*, *Streamer*, *stream()*, *wait()*} as illustrated in Figure 4. We will demonstrate later that the minimal, complete change impact scope is a subset of these accessed model elements.

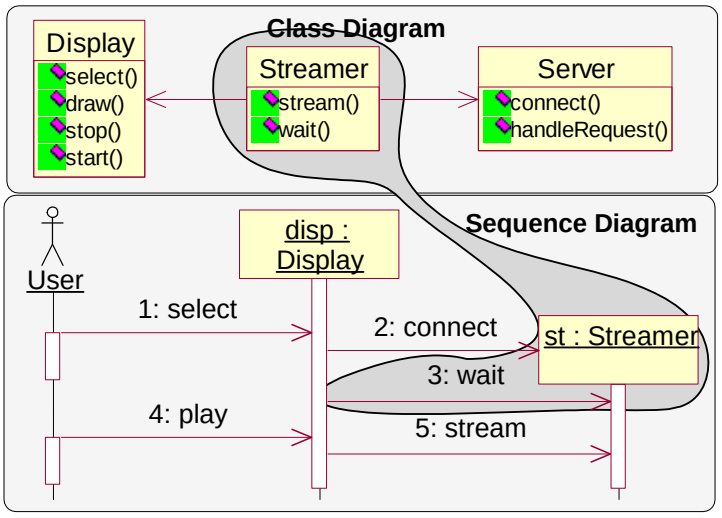

Figure 4. Scope for Message *Wait* evaluated by Consistency Rule 1

Recall from II.A that a consistency rule is typically written from the point of view of a context element (a type of model element) from where its evaluation starts. For consistency rule 1, the context element is an UML Message. Since there are five such messages in the sequence diagram in Figure 4, consistency rule 1 must be evaluated five times – once for each message. To distinguish these evaluations, we define a <consistency rule, model element> pair as a **consistency rule instance, or short CRI.**

$$\boldsymbol{Consistency\ Rule\ Instance} = < ConsistencyRule, ModelElement >$$

$$\boldsymbol{where\ Model\ Element\ instanceOf\ ContextElement}(\boldsymbol{ConsistencyRule})$$

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 returns 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, it follows that solely these accessed model elements are needed to compute their truth values. Or in reverse, a model element that was not accessed during a CRI's evaluation cannot have contributed to its truth value.

If a model element changes then all those CRIs have to be re-evaluated that accessed the changed model element. We thus define the **change impact scope**, or short **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 re-evaluated) that include the changed element in their respective scopes (we discuss this further in III.C). For example, if method *wait* is renamed then the CRI <rule1, wait> needs to be evaluated whereas CRI <rule1, play> need not (because it did not previously access the changed method name).

$$\boldsymbol{Scope}(\boldsymbol{CRI}) = \boldsymbol{Set\ of} < ModelElement, Field > \boldsymbol{accessed\ during\ Evaluation\ of\ CRI}$$

$$\boldsymbol{AffectedCRIs}(\boldsymbol{Change}) = \{\boldsymbol{CRI} \in \boldsymbol{CRIs} \mid \boldsymbol{CRI.Scope\ contains\ Change}\}$$

It is important to note that our change impact scopes are in fact quite small (a fact that will be supported though extensive empirical evidence in Section V). In Section II.C, we identified the

type-base scope for consistency rule 1 as including UML Message, Object, Class, and Operation. However, the instance-based scope *does not* include all instances of these types. For example, message *connect*, object *disp*, class *Server*, or operation *draw* are all not included in the scope of <rule 1, *wait*>. This is why the type-based scope to consistency checking did not scale (by not distinguishing between the types of model elements and its instances, it was not possible to separate what model elements in particular caused the re-evaluation of consistency rules). Our instance-based change impact scope is more precise than the type-base scope. Indeed, we will see later that our approach is not only instant but its performance is also not affected by the size of the model (an important scalability factor).

It is also important to note that consistency rules typically access few, selected fields of model elements only and it is quite possible that two consistency rules access some of the same model elements yet 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”

B. 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 re-evaluate 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 [22]. Some even tried to define explicit change impact rules that complement consistency rules in identifying when and how changes impact a model [5, 21]. However, these approaches require extensive manual effort and are not guaranteed to be correct.

We, on the other hand, do not compute the scopes of CRIs manually. Instead, we observe their behavior in the form of model profiling. Today, profilers are used extensively on source code to observe what code is executed when, how often, 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, how often, and in what order *during consistency checking*. This profiler is based on an infrastructure we developed over several years (Section IV) which allows us to observe how models are used and changed over time. To date, we implemented this infrastructure on the commercial modeling tools IBM Rational Rose™, Matlab/Stateflow™, the Eclipse-based IBM Rational Software Modeler™, and the Dopler Product Line modeling tool. The model profiling technology is discussed in detail in [12]. 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. The questions are: (1) is this scope complete and (2) is this scope small (ideally minimal). These two questions are answered next:

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 II.A). 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 (i.e., because it is stateless, all data must come from the model that is being profiled).

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) functions: =, <, >, similar, etc.	elements in path1 and path2 in scope
condition	function(path1) functions: size, count, existence	elements in path1 in scope
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

Figure 5. Accessible and Accessed Model Elements

The completeness issue is obvious for simple, unconditional statements where a consistency rules navigates a set of model elements. For example, consistency rule 1 first identified all 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). Consistency rules rely on such path expressions for locating model elements but they also rely on a range of first-order logic for further processing. Figure 5 lists common constructs found in consistency rules [25] 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 potentially accessible by a consistency rule.

Take, for example, the OR operator. The OR operator requires a condition to be evaluated only until the first sub-condition is true. For example, if *A* is true in *A or B* then *B* is not evaluated. Only if *A* is false then *B* is 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 a potentially accessible element that by chance was not accessed and, consequently, not all accessible model elements are accessed during a rule's evaluation and its scope appears incomplete. Fortunately, this level of completeness is not required for our 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 expressions's truth value. Thus, only *A* need be in the scope. The outcome of the evaluation cannot change if a model element is changed that is potentially accessible but was not accessed. In other words, for as long as *A* remains true in *A or B*, changes to *B* do not matter and are not required to be in the scope.

We encounter a similar situation with AND conditions because there a condition must be evaluated only until the first subcondition is false to make the entire condition false (e.g., if *A* is false in *A and B* then *B* need not be evaluated and need not be in the scope). Again, not all potentially accessible model elements are accessed if the condition evaluates to false but again this level of completeness is not required. In addition to the OR and AND constructs, there are other constructs where not all accessible model elements are accessed (e.g., for all, exists quantifications, implies) but even there completeness only requires accessed elements and not potentially accessible elements.

The scope determined by our approach is thus complete because it contains at least the model elements needed for its evaluation. This scope is complete even though it does not contain all potentially accessible model elements.

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 re-evaluation that does not change a consistency rule's truth value is unnecessary because it re-computes what is already known. We believe that it is infeasible to compute a minimal scope because such a scope depends on the current state of the model and its potential changes. To illustrate this, consider once again the CRI <rule1, wait>. We know that its scope must include the message *wait*. Yet consider a message name change from *wait* to *stream*. While this change is alike the previously discussed change from *wait* to *suspend*, this change is different in that it does not affect the truth value because there is a corresponding method *stream*() in the base class *Streamer*.

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 still computationally scalable and Section V presents the empirical evidence based on 34 models, 435,932 unique <model element, field> pairs (=scope elements), and over 290,826 unique CRIs. We found that the scope sizes, while not minimal, were small in including 21 model elements or fewer for 95% of all CRIs. But most significantly, we found that the scope sizes did not increase with model sizes. They stayed constant. This explains why 95% of all model changes required 7ms or less evaluation time – with a worst case of less than 2 seconds.

This is not to say that the designer or tool builder 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 re-evaluating the consistency of a model change. To illustrate this, consider the two different, though equivalent implementation choices for consistency rule 1 in Figure 6. The top implementation identifies all operations in the receiver's base class and all its parent classes. It then retrieves the names of all these operations (operations->name) and finally checks whether the message name is contained among the operation names. The second implementation retrieves the base class only and first iterates over the methods of that base class. If a suitable method is found then the algorithm terminates. Otherwise it continues to search the operations of 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, because it likely has a smaller scope than the first implementation, it is less often re-evaluated (smaller scope implies fewer re-evaluations).

Implementation Choice 1 for Consistency Rule 1:

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

Implementation Choice 2 for Consistency Rule 1:

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

Figure 6. Two behaviorally-different implementation of Consistency Rule 1 that are equivalent in terms of the results produced

Since both implementations are equivalent (i.e., they correctly implement consistency rule 1) how are we to interpret their differences in scope? The answer to this is similar to the discussion on the AND/OR constructs 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 more potentially accessible model elements than the second implementation. Both implementations include all necessary change impact scope elements and both implementations are thus valid – albeit the first implementation is less efficient than the other.

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 Figure 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 if and only if a model element in its scope changes. Thus, the re-evaluation of a rule's scope coincides with the evaluation of its truth value. This also implies that the scope of affected CRIs must be re-computed every time one of their model elements change but this overhead is negligible (i.e., the <7ms evaluation time for 95% of all changes already includes this overhead).

Summary

In summary, the scope of a consistency rule cannot be computed through static analysis. However, we demonstrated that through the use of a model profiler, the scope can be observed – even for models within commercial modeling tools (e.g. IBM Rational Rose). The observed scope is automatically computed and the overhead of computing it is very small. While, the scope does consume memory, it is typically small and bounded and it does not increase with the size of the model. However, we did find that the evaluation time increases linearly with the number of consistency rules. This is a known scalability issue of consistency checking and discussed in more detail later.

C. Recognizing Consistency Rule Instances, Creating, and Destroying Them

Instant consistency checking requires an understanding when, where, and how the model changes. For this purpose, our approach monitors the designer while using the modeling tool. Figure 7 shows the architecture of our approach. It depicts the modeling tool on 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., an UML design model).

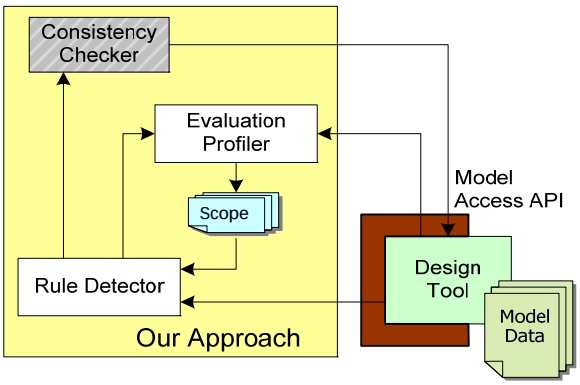

Figure 7. Architecture for Our Approach

The *Consistency Checker* (top-left) accesses the design model and while doing so the *Evaluation 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 what CRIs accessed them. This 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:

```
RuleDetector(changedElement, scope)
  for every CRI where scope contains changedElement
    evaluate <rule, changedElement>
  end for
```

There is an obvious chicken-and-the-egg problem here. The rule detector requires the scope database to determine what rules to re-evaluate 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 to 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.

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 an UML message. If there is no message then there is no need to evaluate consistency rule 1. If there are multiple messages (as in Figure 1) then consistency rule 1 must be evaluated multiple times (once per message). Our approach simply creates CRIs when their context elements are created and it destroys CRIs once their context elements are destroyed.

The modified *RuleDetector* algorithm below illustrates rule creation, destruction, and re-evaluation 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 already 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 ruleInstance where ruleInstance.rootElement=changedElement
```

```
      destroy <ruleInstance, changedElement>
    end for
  end if
  for every ruleInstance where ruleInstance.scope contains changedElement
    evaluate <ruleInstance, changedElement>
  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 meta model 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.

D. Instant and Lazy Consistency Checking

The *RuleDetector* algorithm shown above is not optimal since it causes certain CRIs to be re-evaluated more often than needed. We discussed in II.B 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. In II.B, we showed an example where the simple creation of a message caused three changes: the creation of the message and the modification of the two objects who act as sender and receiver for the message. The deletion of a method, for example, destroys not only the method but it also modifies the class who owns the method because the class now has one less method available. Even a simple modification, say, moving a dependency from one class to another, causes multiple change notifications because in addition to the dependency, the two classes are modified – the class that used to have the dependency and the class that now has the dependency.

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 process of a change – say, where the dependency has been removed from one class but not yet added to the other. Furthermore, it is also not useful to re-evaluate a CRI more than once per change group.

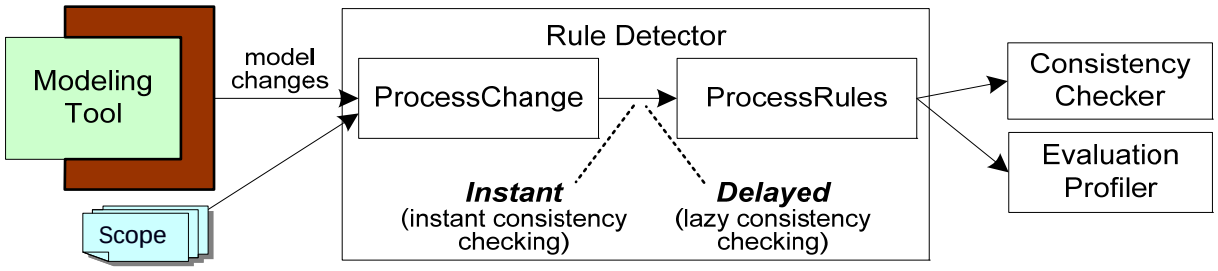

Figure 8. Filtering CRIs in the Rule Detector to Support Faster Instant and Lazy Consistency Checking

The *RuleDetector* algorithm discussed above investigated changes instantly and independently of one another. It thus triggers multiple re-evaluations 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. Figure 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, a designer may wish to make a sequence of changes before re-evaluating the consistency of these changes. This is particularly then useful when changes are explorative and the designer knows that the changes are likely to conflict with the rest of the model. In such cases, the designer may not care to receive intermittent inconsistency feedback but rather desires feedback at the end, after the sequence of changes is completed. This is

commonly referred to as lazy consistency checking where a designer may choose to delay consistency checking (i.e., every hour, when the designer says so, before the designer checks in the model, etc.).

E. 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 designers to fix them instantly. Our approach tracks all presently-known inconsistencies and lets the designers explore inconsistencies according to their interests in the model. If a designer 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, 14], we discuss how our technology helps the designer resolve inconsistencies at some later time. This issue is out of the scope of this paper.

IV. MODEL/ANALYZER TOOL

The Model/Analyzer tool implements our instant consistency checking infrastructure (Figure 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 [8]. The first two implementations are based on top of the UML13 infrastructure [11], the third implementation is based on the EMF (Eclipse Modeling Framework), and the forth implementation is a non-standardized 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

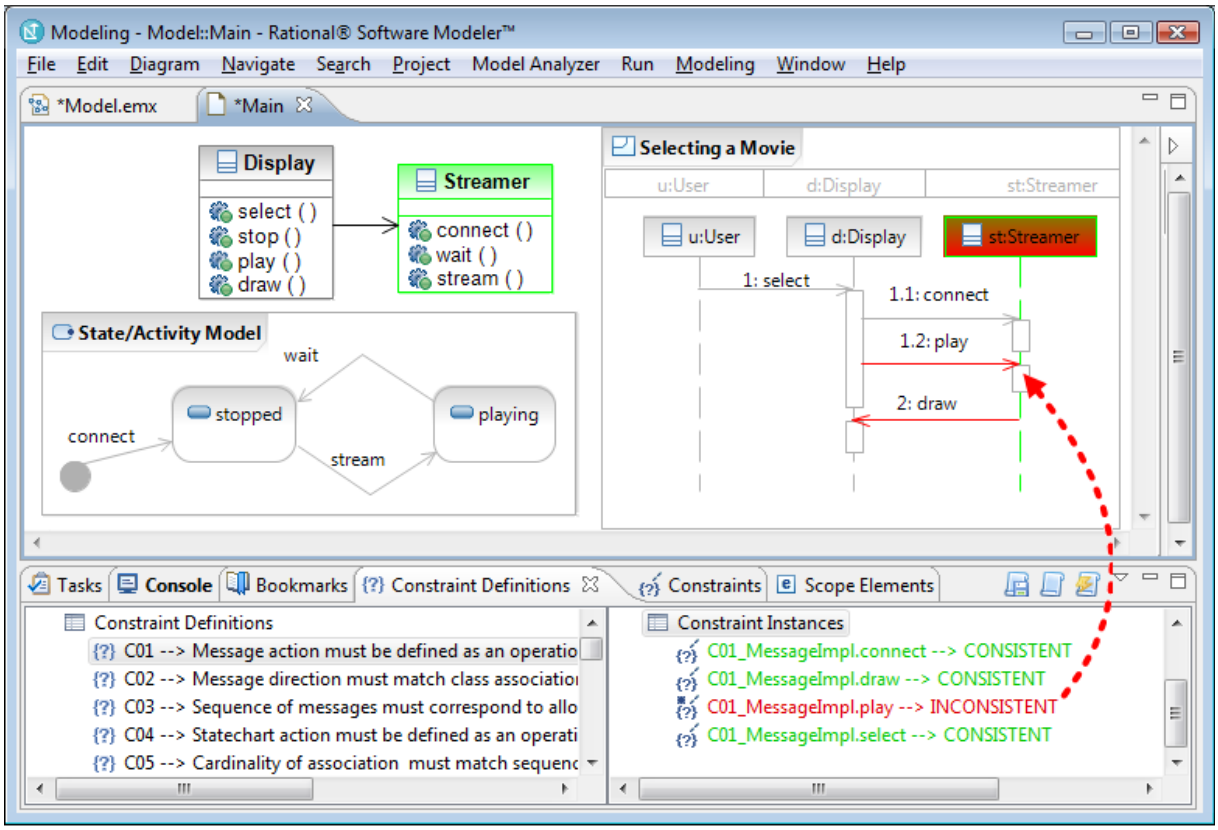

Figure 9. Model/Analyzer Tool Depicting an Inconsistency in IBM Rational Software Modeler™

we have implemented consistency rules in Java, J#, C#, and OCL (Object Constraint Language [35]) at this point. Figure 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 in green. 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 bi-directional navigation is essential for understanding and resolving inconsistencies.

This tool essentially automates all the difficulties of instant consistency checking discussed in

this paper and it was used for the empirical evaluation discussed in Section V. To include new consistency rules, the tool provides an extension point for adding/removing consistency rule at any point in the design process.

V. VALIDATION

Instant consistency checking is only then 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 (Figure 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.

Figure 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 to very large models although it was close to instant for medium-sized models. And it showed that our approach was not affected by the model size at all. This data was 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. Figure 10 depicts the evaluation time across this large set of changes. We see that the evaluation time was less than 7ms for 95% of all model changes – with a median on 0.2ms. We consider an evaluation time of 100ms 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 2.2GHz.

Table 1. Study Models used for Empirical Evaluation[1]

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

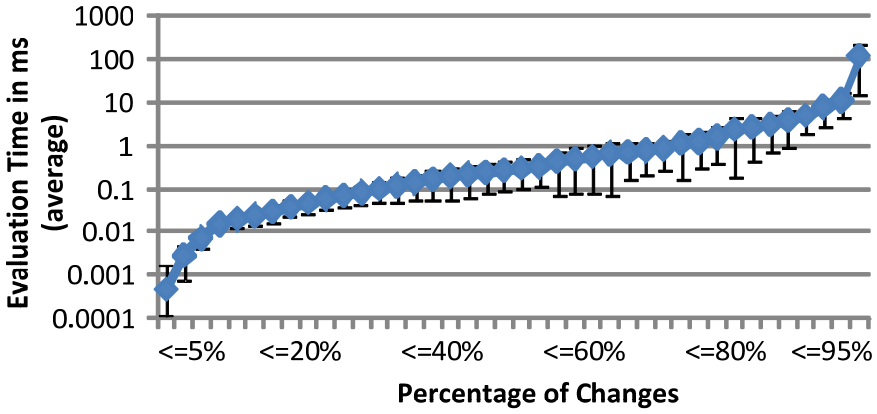

Figure 10. Evaluation Time in ms for Model Changes

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. 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 re-computation option is a one-time expense upon loading. The persistent storage option is also a one-time expense and reasonable as the scope database grows linearly with the size of the model only. We will first discuss the computation cost of loading and incremental re-evaluation. 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 open 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.

A. 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.

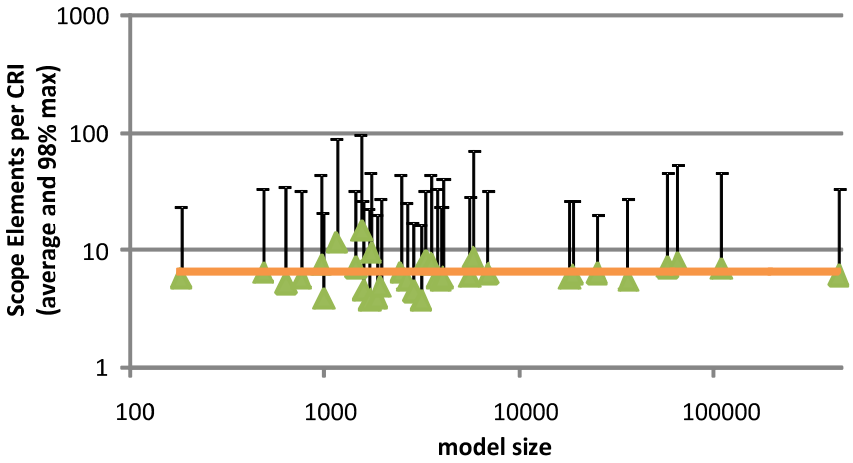

Figure 11. Scope Sizes of CRIs remain constant with the Model Size

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. Figure 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% maximum) where the solid dots are the

average values for any given model. Notice the constant, horizontal line of average scope 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(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. Figure 12 depicts the average *ACRI* through solid dots and the 98% maximums.

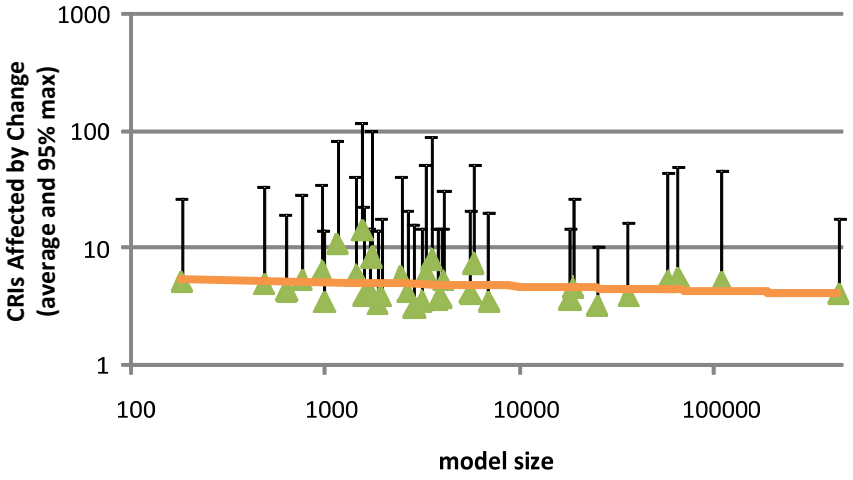

Figure 12. Consistency Rule Instances (CRIs) affected by a Change

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 3-11 depending on the model).

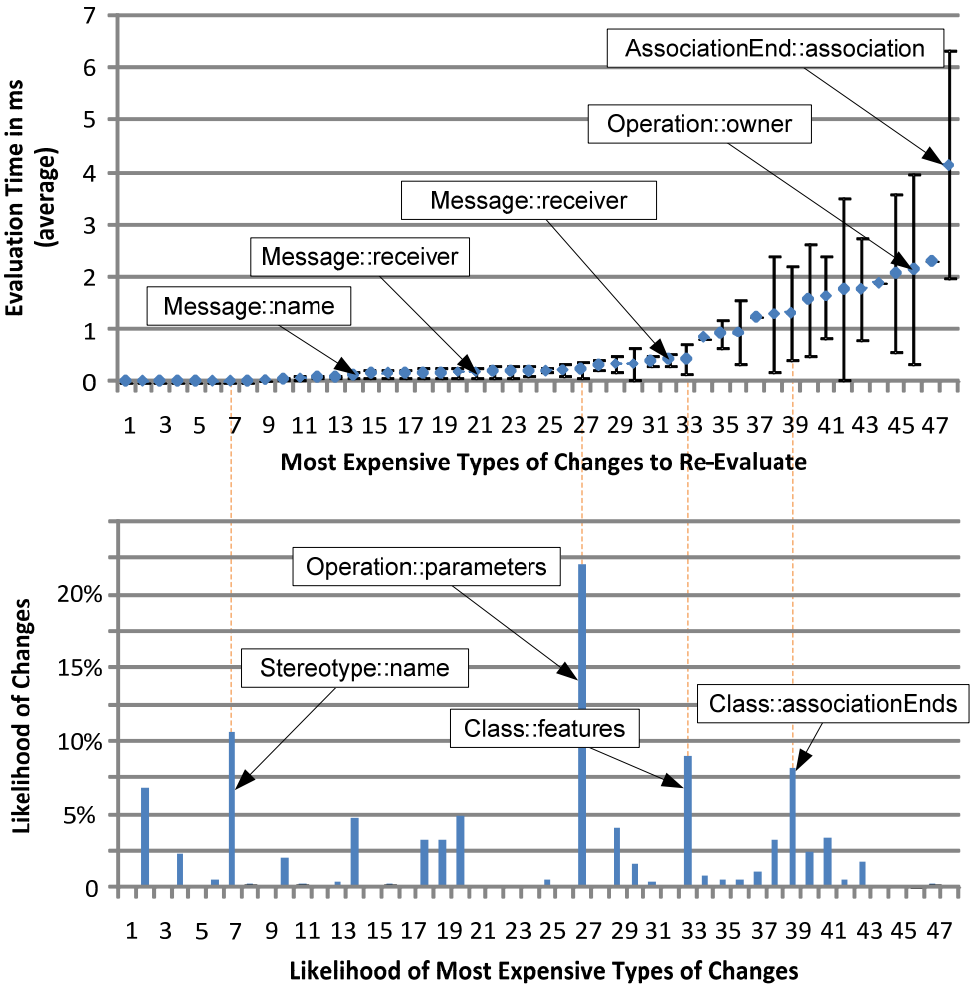

Figure 13. Most Expensive Types of model Changes and Likelihood of these Changes Occuring

Figure 13 depicts the average cost of evaluating a model change based on the type of change (top). We see that a change to the *association* field of an *AssociationEnd* was the most expensive kind of change with over 4ms re-evaluation cost in average. A message name change (as was used several times in this paper) was comparatively cheap with 0.12ms to re-evaluate in average. First and foremost, we note that all types of model changes are quite reasonable expensive to re-evaluate. This implies that irrespective of how often certain types of change happen, our approach performs 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 4075 changes across them. Figure 13 (bottom) depicts that 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 re-evaluation time and that there are very rare outliers (0.00011% of changes with evaluation time >100ms). The reason for this is obvious in Figure 14 where we see that it is exponentially unlikely for CRIs to have larger scope sizes (top) or for changes to affect many CRIs (bottom). We show this data to exemplify how similar the 34 models are in that regard even though these models are vastly different in size, complexity, and domain.

Figure 14 (top) depicts for all 34 models separately what percentage of CRIs (y-axis) had a scope of <= 5, 10, 15,… scope elements (x-axis). The table shows that over 95% of all CRIs accessed less than 15 fields of model elements (scope elements). Figure 14 (bottom) depicts for all 34 models separately what percentage of changes (y-axis) affected <= 2, 4, 6, … CRIs. The table shows that 95% of all changes affected fewer than 10 CRIs (*ACRI*).

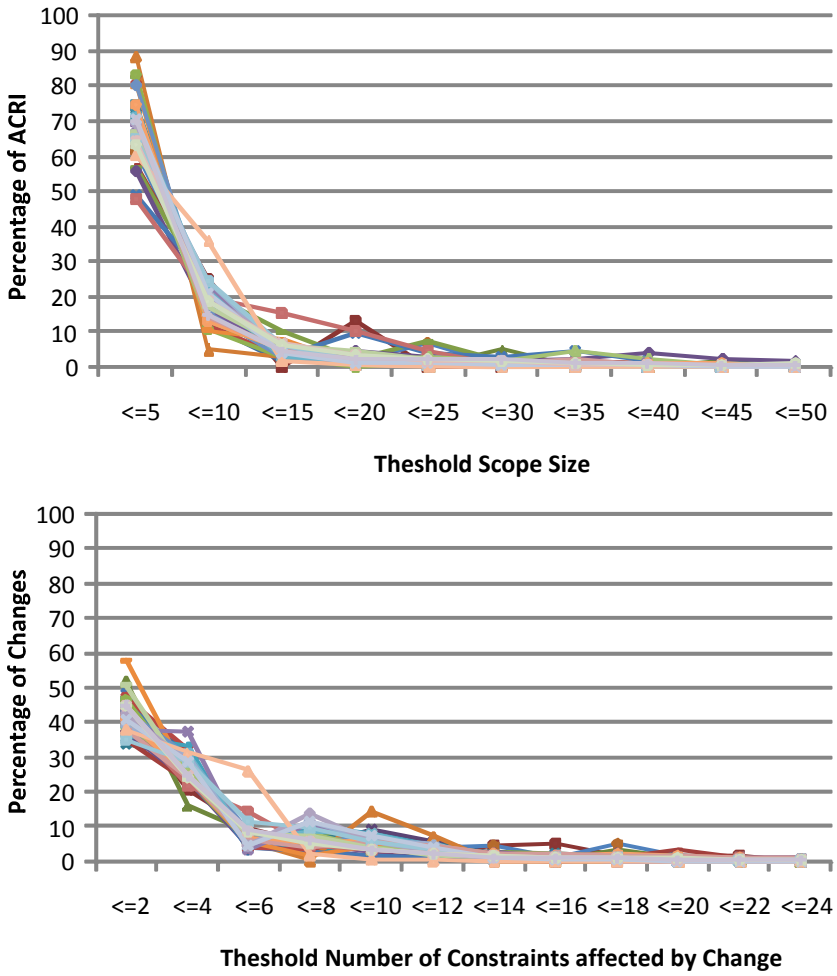

Figure 14. Number of Model Elements Accessed by Constraints

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 rules typically are not global; they are in fact surprisingly local in their investigations. This is demonstrated in Figure 15 which depicts the cost of evaluating changes for each consistency rule separately. Still, each consistency rule takes time to evaluate

and Figure 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 in average 0.004-0.21ms 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 [35] were comparative expensive due to the high cost of interpreting them.

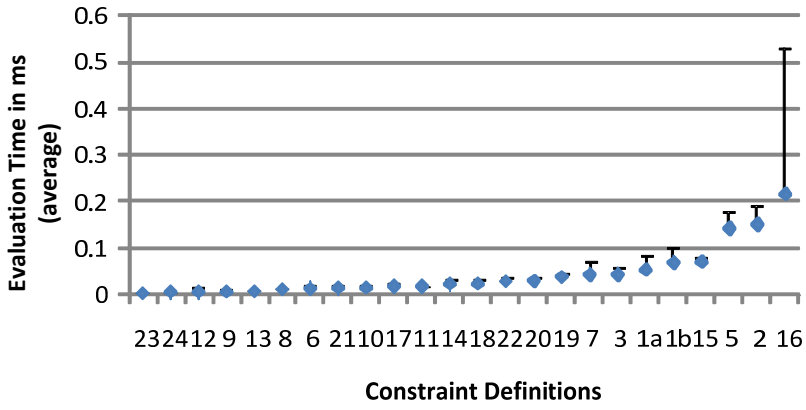

Figure 15. Cost of Adding a Consistency Rule

B. Memory Cost

On the downside, our approach does require additional memory for storing the scopes. Figure 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. As with the evaluation time, this cost also increases with the number of consistency rules (*RT#*). The memory cost is RT# * S_{size}.

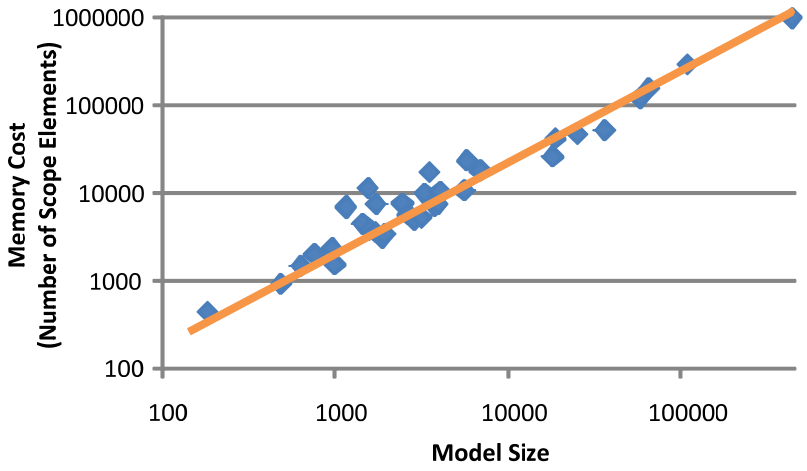

Figure 16. Memory Cost Increases Linearly with Model Size

C. 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 having implemented our approach on UML 1.3, UML 2.1, Matlab/Stateflow, and Dopler Product Line, and having used 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.

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 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 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 scales also.

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 constraint rules covered all 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 overlap mostly with other UML diagrams and thus not affect our rules much. This implies that more consistency rules do not necessarily imply a longer evaluation time. However, given that all consistency rules evaluated added less than 1ms each to the total evaluation time, we do not foresee scalability issues even with an order of magnitude larger RT#.

VI. 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 to some intermediate representation to compare there.

For example, Tsiolakis-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-Lemus [2] also follow along the tracks of VisualSpecs in their use of a formal language to substitute statechart and dataflow diagrams. Streaten et al. [20] explore the use of description logic to detect inconsistencies between sequence and statechart diagrams. Campbell et al. [6] make use of the SPIN Model checker to evaluate a range of consistency problems within and across UML diagrams. Or, Zisman and Kozlenkov [37] 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 any 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 exist 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 [32] 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 user of an intermediate representation is not a pre-requisite for consistency checking. Indeed, it is possible to write consistency rule that directly compare design models rather than

transforming them first [17, 18, 24, 28]. However, the most interesting ones among these are the xLinkIt [24] and ArgoUML [28] approaches because they are capable of performing incremental consistency checking.

xLinkIt [24] is a 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 [27] 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 makes his approach certainly more incremental. However, the incremental updates in his study suggest non-instant 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 are not.

ArgoUML also detects inconsistencies in UML models [28] 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% 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 those consistency rules are evaluated that are affected by that element's type. We will demonstrate 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 non-intrusive 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 instant as our approach.

Blanc et al. [3] approach the issue of incremental consistency checking from the perspective of model changes. This is in principle alike 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 guarantees 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" [1, 16] 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 non-trivial problem because the scope of an (in)consistencies is continuously affected by model changes. Our approach could also be used for lazy consistency checking which has been explored in [29] 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 necessary eventually. This issue is explored in [13, 14, 25, 36].

In a broader sense, current approaches to consistency checking borrow from programming environments such as Centaur or Gandalf [19] that incrementally evaluate syntactic or even semantic [15] 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 do not use a grammar-based language, or formal language for that matter. Yet, these issues are of interest here because they also investigate decentralized consistency checking [20] and consistency checking among different languages which is considered outside the scope here [33].

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 [23] defines a mapping between choices and the constraints they affect. Constraints are only then re-evaluated 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 thus 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.

Viewpoints [9] 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 on “upstream” modeling techniques and it addresses issues such as how to resolve inconsistencies [13, 25, 26, 31] 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.

VII. CONCLUSIONS

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 29 large-scale models that the average model change cost 1.4ms, 98% of the model changes cost less than 7ms, 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 re-evaluate the rule. In the case of an inconsistency, this scope tells the engineer all 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 for 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 non-scalable fashion although it must be stressed that of the hundreds of rules known to us, none fall into this category.

ACKNOWLEDGMENT

We would like to thank Iris Groher and Alexander Reder in porting the UML/Analyzer tool, based on IBM Rational Rose™, to the Eclipse-based IBM Rational Software Modeler™.

REFERENCES

[1] R. Balzer, "Tolerating Inconsistency, " *Proceedings of 13th International Conference on Software Engineering (ICSE)*, 1991, pp. 158-165.

[2] B. Belkhouche, and C. Lemus, "Multiple View Analysis and Design, " *Proceedings of the Viewpoint 96: International Workshop on Multiple Perspectives in Software Development*, 1996.

[3] X. Blanc, I. Mounier, A. Mougenot, and T. Mens, "Detecting Model Inconsistency through Operation-Based Model Construction, " *30th International Conference on Software Engineering (ICSE)*, Leipzig, Germany, 2008, pp. 511-520.

[4] 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*, New Jersey: Prentice Hall, 2000.

[5] L. C. Briand, Y. Labiche, and L. O'Sullivan, "Impact Analysis and Change Management of Uml Models, " *Proceedings of the International Conference on Software Maintenance (ICSM)*, Amsterdam, The Netherlands, 2003, p. 256.

[6] L. A. Campbell, B. H. C. Cheng, W. E. McUmber, and K. Stirewalt, "Automatically Detecting and Visualising Errors in Uml Diagrams," *Requirements Engineering Journal*, vol. 7(4), pp. 264-287, 2002.

[7] B. H. C. Cheng, E. Y. Wang, and R. H. Bourdeau, "A Graphical Environment for Formally Developing Object-Oriented Software, " *Proceedings of 6th International Conference on Tools with Artificial Intelligence*, New Orleans, USA, 1994, pp. 26-32.

[8] D. Dhungana, R. Rabiser, P. Grünbacher, K. Lehner, and C. Federspiel, " Dopler: An Adaptable Tool Suite for Product Line Engineering, " *11th International Software Product Line Conference (SPLC 2007), Proceedings: The Second Volume*, Kyoto, Japan, 2007, pp. 151-152.

[9] S. Easterbrook, and B. Nuseibeh, "Using Viewpoints for Inconsistency Management," *IEE Software Engineering Journal*, vol. 11(1), pp. 31-43, 1995.

[10] A. Egyed, "Instant Consistency Checking for the Uml, " *Proceedings of the 28th International Conference on Software Engineering (ICSE)* Shanghai, China, 2006, pp. 381-390.

[11] A. Egyed, and B. Balzer, "Integrating Cots Software into Systems through Instrumentation and Reasoning," *International Journal of Automated Software Engineering (JASE)*, vol. 13(1), pp. 41-64, 2006.

[12] A. Egyed, and R. Balzer, "Integrating Cots Software into Systems through Instrumentation and Reasoning," *Automated Software Engineering* vol. 13(1), pp. 41-64, 2006.

[13] A. Egyed, "Fixing Inconsistencies in Uml Design Models, " *Proceedings of the 29th International Conference on Software Engineering* 2007, pp. 292-301.

[14] A. Egyed, E. Letier, and A. Finkelstein, "Generating and Evaluating Choices for Fixing Inconsistencies in Uml Design Models, " *Proceedings of the 23rd International Conference on Automated Software Engineering (ASE)*, L'Aquila, Italy, 2008.

[15] W. Emmerich, "Gtsl - an Object-Oriented Language for Specication of Syntax Directed Tools, " *Proceedings of the 8th International Workshop on Software Specification and Design*, Velen, Germany, 1996, pp. 26-35.
[16] S. Fickas, M. Feather, and J. Kramer, *Proceedings of Icse-97 Workshop on Living with Inconsistency*, Boston, USA, 1997.
[17] A. Finkelstein, D. Gabbay, A. Hunter, J. Kramer, and B. Nuseibeh, "Inconsistency Handling in Multi-Perspective Specifications," *Transactions on Software Engineering (TSE)*, vol. 20(8), pp. 569-578, 1994.
[18] J. Grundy, J. Hosking, and R. Mugridge, "Inconsistency Management for Multiple-View Software Development Environments," *IEEE Transactions on Software Engineering (TSE)*, vol. 24(11), 1998.
[19] A. N. Habermann, and D. Notkin, "Gandalf: Software Development Environments," *IEEE Transactions on Software Engineering (TSE)*, vol. 12(12), pp. 1117, 1986.
[20] S. M. Kaplan, and G. E. Kaiser, "Incremental Attribute Evaluation in Distributed Language-Based Environments, " *Proceedings of the 5th Annual Symposium on Principles of Distributed Computing*, Calgary, Canada, 1986, pp. 121-130.
[21] M. Lee, A. J. Offutt, and R. T. Alexander, "Algorithmic Analysis of the Impacts of Changes to Object-Oriented Software, " *34th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS)*, Santa Barbara, CA, USA, 2000, pp. 61-70.
[22] M. Lindvall, and K. Sandahl, "Practical Implications of Traceability," *Journal on Software - Practice and Experience (SPE)*, vol. 26(10), pp. 1161-1180, 1996.
[23] A. K. Mackworth, "Consistency in Networks of Relations," *Journal of Artificial Intelligence*, vol. 8(1), pp. 99-118, 1977.
[24] C. Nentwich, L. Capra, W. Emmerich, and A. Finkelstein, "Xlinkit: A Consistency Checking and Smart Link Generation Service," *ACM Transactions on Internet Technology (TOIT)*, vol. 2(2), pp. 151-185, 2002.
[25] C. Nentwich, W. Emmerich, and A. Finkelstein, "Consistency Management with Repair Actions, " *Proceedings of the 25th International Conference on Software Engineering (ICSE)*, Portland, Oregon, USA, 2003, pp. 455-464.
[26] B. Nuseibeh, and A. Russo, "On the Consequences of Acting in the Presence of Inconsistency, " *Proceedings of the 9th International Workshop on Software Specification & Design*, Ise-Shima, Japan, 1998, pp. 156-158.
[27] S. Reiss, "Incremental Maintenance of Software Artifacts," *IEEE Transactions on Software Engineering*, vol. 32(9), pp. 682-697, 2006.
[28] J. Robins, and others, "Argouml," *http://argouml.tigris.org/*.
[29] N. Roussopoulos, "An Incremental Access Method for Viewcache: Concept, Algorithms, and Cost Analysis," *ACM Transactions On Database Systems*, vol. 16(3), pp. 535-563, 1991.
[30] J. Rumbaugh, J. Ivar, and B. Grady, *The Unified Modeling Language Reference Manual*: Addison Wesley, 1999.
[31] M. Sabetzadeh, S. Nejati, S. Liaskos, S. Easterbrook, and M. Chechik, "Consistency Checking of Conceptual Models Via Model Merging, " *Proceedings of the 15th IEEE International Requirements Engineering Conference (RE)*, New Delhi, India, 2007.
[32] W. Shen, K. Wang, and A. Egyed, "An Efficient and Scalable Approach to Correct Class Model Refinement," *IEEE Transactions on Software Engineering*, vol. 35, 2009.
[33] 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, " *Proceedings of the 4th Symposium on Software Development Environments*, Irvine, CA, 1998.
[34] A. Tsiolakis, and H. Ehrig, "Consistency Analysis of Uml Class and Sequence Diagrams Using Attributed Graph Grammars, " *Proceedings of Graph Transformation & Graph Grammars (GRATA)*, Berlin, Germany, 2000, pp. 77-86.
[35] J. Warmer, and A. Kleppe, *The Object Constraint Language*: Pearson Education, 2003.
[36] Y. Xiong, Z. Hu, H. Zhao, H. Song, M. Takeichi, and H. Mei, "Supporting Automatic Model Inconsistency Fixing., " *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*, Amsterdam, The Netherlands, 2009.
[37] A. Zisman, and A. Kozlenkov, "Knowledge Base Approach to Consistency Management of Uml Specification, " *16th IEEE International Conference on Automated Software Engineering (ASE)*, San Diego, USA, 2001, pp. 359-363.