Instant Consistency Checking for the UML

Alexander Egyed
Teknowledge Corporation
4640 Admiralty Way, Suite 1010
Marina Del Rey, CA 90292, USA
aegyed@teknowledge.com

ABSTRACT

Inconsistencies in design models should be detected immediately to save the engineer from unnecessary rework. Yet, tools are not capable of keeping up with the engineers' rate of model changes. This paper presents an approach for *quickly, correctly, and automatically* deciding what consistency rules to evaluate when a model changes. The approach does not require consistency rules with special annotations. Instead, it treats consistency rules as black-box entities and observes their behavior during their evaluation to identify what model elements they access. The UML/Analyzer tool, integrated with IBM Rational Rose™, fully implements this approach. It was used to evaluate 29 models with tens-of-thousands of model elements, evaluated on 24 types of consistency rules over 140,000 times. We found that the approach provided design feedback correctly and required, in average, less than 9ms evaluation time per model change with a worst case of less than 2 seconds at the expense of a linearly increasing memory need. This is a significant improvement over the state-of-the-art.

Categories and Subject Descriptors

D.2.10 [**Design**]: Validation

General Terms

Measurement, Documentation, Design, Languages, Verification.

Keywords

Incremental Analysis, Consistency, Design Feedback.

1. INTRODUCTION

The UML [19] is a collection of loosely-connected, diagram-centric design notations. It allows engineers to explore the design of a software system through independent views. However, our work with engineers from industry (e.g., Boeing Company, SwRI) has shown that it is all too easy to make contradictory design decisions and that it is hard to recognize these *inconsistencies*.

After decades of progress on model consistency [8] and consistency rules [18], this revelation is disturbing. Some are quick to blame industry for its failure to adopt existing solutions. Yet, we observed that the problem was not the lack of consistency checking but the timeliness and quality of its feedback. Consistency checking on the larger models took hours and was thus performed occasionally only. At the end, the engineers were presented with many inconsistencies (the worst industrial model we investigated contained 1650 inconsistencies) but were not told all the model elements that were involved in any given inconsistency (needed for fixing it or living with it [7]). To fix the inconsistencies, engineers then had to interrupt their work flow and recollect all the concerns that had affected the inconsistencies made hours, days, or weeks earlier. But the engineers were most frustrated by also having to revisit and change follow-on decisions that were based on inconsistent data.

The engineers' frustration is easy to understand. It used to be common for programmers to be presented with syntax and semantic errors in their algorithms *after* the fact. Today, programmers benefit from instant compilation and programming environments point out many (if not all) syntax and semantic errors within seconds of making them – usually in a non-intrusive manner. *Instant error feedback of any kind is a fundamental best practice in the software engineering process*.

This paper presents an approach to the instant consistency checking of UML models, which is an adaptation to incremental consistency checking. While incremental consistency checking is typically much faster than batch consistency checking, it is not necessarily *instant* [13]. Moreover, most techniques for incremental consistency checking require consistency rules *with additional declarations* [15].

We demonstrate that our approach keeps up with an engineer's rate of model changes, even for very large-scale industrial models with tens of thousands of model elements. Furthermore, our approach treats consistency rules given by the engineers as black boxes (no *special declarations* are required) and works together with existing commercial modeling tools such as IBM Rational Rose™ (no *special modeling tools* are required). Yet, 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 profiler for consistency checking. The profiling data is used to establish a correlation among model elements, consistency rules, and inconsistencies. Based on this correlation, *we can decide when to re-evaluate consistency rules* and *when to display inconsistencies* - allowing the engineers to quickly identify all inconsistencies that pertain to any part of the model of interest at any time (i.e., living with inconsistencies [7]).

Even though our approach (or any approach) is not guaranteed to be instant for any consistency rule, this paper presents empirical evidence that our approach is instant for the 24 consistency rules we studied (these rules were chosen based on the needs of our industrial partners). The evaluation showed that 99% of the model changes were evaluated within less than 50ms each (with an average of 9ms), even on the largest industrial models we had available. It is truly instant. On the downside, our approach required additional memory for storing the profiling data – a memory need that rose linearly with the size of the model and the number of consistency rules.

While the tool and its evaluation were based on the UML 1.3 notation, we believe that the infrastructure applies equally to other modeling languages (i.e., UML 2.0) because every consistency rule has to access model elements and thus can be profiled. *The consistency rules may change but the infrastructure for evaluating them instantly remains the same.* To date, our approach was implemented on top of a concrete consistency rule language, consistency checker, and modeling tool. If a different modeling tool is used then the profiler needs to be customized to that tool and the consistency rules have to be customized to a language/checker available for that tool. Doing so does not necessarily require access to the source code of the modeling tool or the consistency checker.

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. These issues are deferred to future work due to space limitations.

2. 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, usually formal, representation to compare there.

For example, Tsiolakis-Ehrig [18] 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. Using an intermediate representation has many advantages. Yet, for instant consistency checking it has the disadvantage of also having to implement instant transformation in addition to instant consistency checking.

While it is important to know about inconsistencies, it is often too distracting to resolve them right away. The notion of “living with inconsistencies” [1,7] 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 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. This is a non-trivial problem because the scope of an (in)consistency is continuously affected by model changes. Our approach may also be used for lazy consistency checking which has been explored in [16] but is out of the scope of this paper.

xLinkIt [13] is a XML-based environment for evaluating the consistency of “documents.” Such documents could be anything including UML design models. 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 is not meant to support instant consistency checking with frequent model changes. Instead, it is most useful for the occasional exchange of models and for enforcing consistency constraints in a uniform manner across different modeling languages.

ArgoUML also detects inconsistencies in UML models [15] 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.

Current approaches to consistency checking also borrow from programming environments such as Centaur or Gandalf [9] that incrementally evaluate syntactic or even semantic [6] 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 grammar-based languages or formal languages. Yet, this work is of interest because it also investigated decentralized consistency checking [11] and consistency checking among different languages [17] which is outside the scope here.

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 [12] 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.” It is thus known, in advance, what choices a constraint will encounter. Consistency rules in UML typically are black-box constraints. This is the main reason why most approaches to incremental consistency checking require additional annotations in the UML domain.

Viewpoints [10] is a classical approach to consistency checking. It emphasizes on “upstream” modeling techniques and it addresses issues such as how to resolve inconsistencies [14] 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.

3. INSTANT CONSISTENCY CHECKING

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.

3.1 Consistency Rules and Illustration

The illustration in Figure 1 depicts three diagrams created with the UML [19] modeling tool IBM Rational Rose™. The given model represents an early design-time snapshot of a real, albeit simplified, 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.

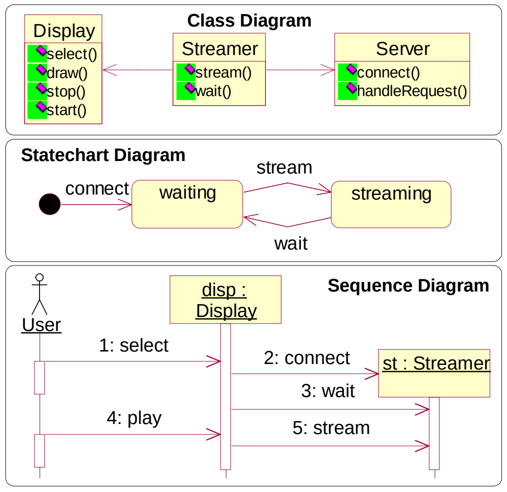


Figure 1. Simplified UML Model of the VOD System

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*.

Consistency rules for UML describe conditions that an UML model must satisfy for it to be considered a valid UML model (e.g., syntactic well-formedness, coherence between different diagrams, and even coherence between different models). Figure 2 describes three such consistency rules on how UML sequence diagrams (objects and messages) relate to class and statechart diagrams. These rules are implemented in a language provided by the UML/Analyzer, a tool discussed later in detail. A consistency rule may be thought of as a *condition* that evaluates a portion of a model to a *truth value*. Note that rules 2 and 3 are not standard UML well-formedness rules but they are examples of coherence rules between two diagrams.

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

Rule 2	**Calling direction of message must match an association**
	in=message.receiver.base.incomingAssociations;
	out=message.sender.base.outgoingAssociations;
	return (in.intersectedWith(out)<>{})

Rule 3	**Sequence of object messages must correspond to events** (definition sketched) startingPoints = find state transitions equal first message name startingPoints->exists(object sequence equal reachable sequence from startingPoint)

Figure 2. Sample Consistency Rules

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 a rule is evaluated on the same portion of the model twice then it will perform the exact same actions and determine the same truth value. In the following, we define a *model element* to be an instance of an UML type. For example, all messages (e.g., *wait*) are model elements of the UML type *Message*; and all objects (e.g., *st*) are model elements of the UML type *Object*. The UML types are defined according to the UML 1.3 notation[1].

Note that consistency rules are typically expressed from a particular point of view to ease their design and maintenance. For example, consistency rule 1 is expressed from the view of a message (i.e., given a message, is it consistent?). We refer to the *message* as the ***root element*** of the rule. Yet, the consistency rules are not only affected by changes to their root elements. One of the most severe problems of incremental consistency checking is in correctly identifying the true scope of model elements that affect the truth value of any given consistency rule.

3.2 Understanding Change

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 typically not the case and not considered here). Instant consistency checking thus requires an understanding of

[1] We used UML 1.3 because of the needs of our partners.

when, where, and how the model changes. Our approach is based on an UML 1.3 infrastructure we previously developed and integrated with IBM Rational Rose™ and other COTS modeling tools [5]. This infrastructure exposes the modeling data of the COTS modeling tool in a UML-compliant fashion. It also employs a sophisticated change detection mechanism. For example, if an engineer (i.e., while using Rose) creates a message between two objects then this change is recognized as:

New model element: 102 UML.Message
Modified model element: 100 UML.Object [outgoingMessages]
Modified model element: 101 UML.Object [incomingMessages]

The first change notification tells about the creation of a model element of UML type *Message* with an id that uniquely identifies the model element. Since a message was created between two existing objects, both these objects are modified in that one now owns a new incoming message and the other a new outgoing message. The infrastructure thus describes changes in terms of their impact on the model but not the actual activities performed by the engineers. For example, an engineer may create a message in Rose by invoking a menu item or by dragging a toolbar icon, yet the result, the creation, is the same.

3.3 The "What Happens If…" Solution

It is intuitive to think of instant consistency checking in terms of *what happens if* a model element changes. 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 instead 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 many other changes that invalidate the consistency rule 1.

Given that there are potentially many ways on how a model change affects consistency rules, it is difficult to identify *all* of them. The issue of correctness is hard to guarantee although such a solution is likely very efficient. While we found it intuitive to think of instant design feedback in terms of "what happens if," we found that it is too complex to correctly support a large rule base.

3.4 The Type-Based Scope to Consistency

Approaches such as the ArgoUML [15] rely on a *type-based scope* for incremental consistency checking. These approaches use the scope to decide when to evaluate a consistency rule. If a model element changes then all those rules are evaluated that include the type of that model element in its scope. Typically, the type-based scope of a consistency rule includes all types of model elements it accesses. 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*}.

In the case of ArgoUML, the type scope must be provided by a human in the form of special annotations to consistency rules. Unfortunately, the UML has a limited number of (meta-level) classes or class:field pairs that could be used as types. Thus, while UML models may increase to arbitrary sizes, their type scope stays constant. Figure 3 depicts the result of empirically evaluating model changes on 24 consistency rules and 29 sample models. It shows that brute-force consistency checking (of all rules) is only scalable for small-sized models with less than 1000 model elements. While type-based consistency checking is scalable for larger models with up to 10,000 model elements, the figure demonstrates that it cannot support instant consistency checking of arbitrary-sized UML models.

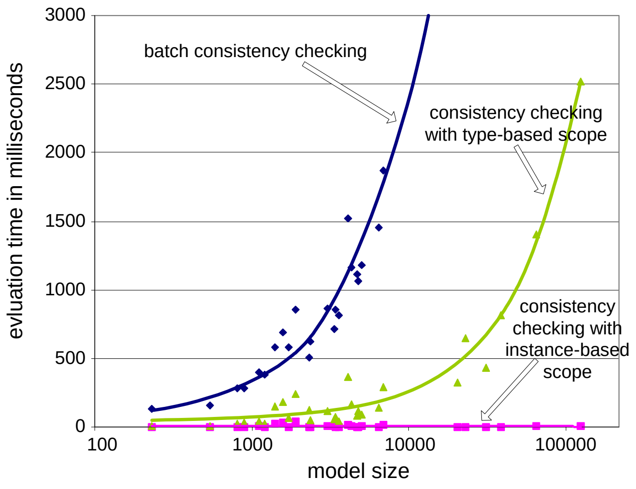


Figure 3. Evaluation Time per Model Change: the evaluation times of batch and type-based scope grow linearly with the size of the model (note the exponential scale of the x-axis) and the instance-based scope stays constant with the size of the model

4. Instance-Based Scope to Consistency

Our approach is similar to the type-based approach to consistency checking. However, instead of using types of model element for the scope, we use the actual instances. Figure 3 shows that an instance-based scope for consistency checking is not only very fast (see the tiny evaluation time) but it is also scalable to arbitrary model sizes (see constant evaluation time regardless of model size). Clearly, an instance-based scope seems ideal, however, it is not possible to predict in advance what model elements (=instances) are accessed by any given consistency rule.

4.1 Rule Types and Instances

During evaluation, a consistency rule requires access to a portion of the model (some of its model elements). We define the accessed portion of a model as the ***scope***. For example, the evaluation of rule 1 on message *wait* first accesses the message, *wait* then the message's receiver object *st*, then its base class *Streamer*, and finally the methods *stream* and *wait* of the base class (recall 3.1). This is how rule 1 was defined in Figure 2. The scope of rule 1 on message *wait* is thus {*wait*, *st*, *Streamer*, *stream()*, *wait()*} as illustrated in Figure 4. This scope includes instances of model elements and not types. The scope of a consistency rule is not constant. For example, the evaluation of rule 1 on message *play* requires access to *play*, *disp* object, *Display* class, and its four methods. Its scope is different from the scope of rule 1 on message *wait* even though both evaluations are based on the same consistency rule (rule type).

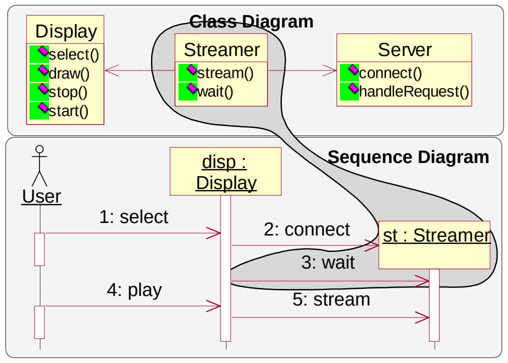


Figure 4. Scope for Message Wait evaluated by Rule 1

We thus must maintain the scope separately for every <rule, root element> pair (e.g., <*rule1*, *wait*>). Recall that a consistency rule is typically written from a particular point of view and starts its evaluation at a particular point – the root element. We define a <rule, root element> pair as a ***rule instance***[2]. A rule instance defines the type of rule it instantiates (e.g., rule 1), the root element (e.g., message *wait*), and its scope. The evaluation of a rule instance accesses all those model elements that are needed to determine its truth. All those model elements are in the scope. All other model elements are not needed. It follows that *a rule instance does not require its evaluation if a model element changes that is not in its scope*. For example, the message *play* is not in the scope of rule instance <*rule1*, *wait*> and its change does not affect this rule instance. However, the object *st* is in the scope of the rule instance and its change *could* affect the truth.

If a model element changes then all rule instances are evaluated that include the changed element in their scopes. For example, if method *wait* is renamed then the rule instances <*rule1*, *connect*>, <*rule1*, *wait*>, and <*rule1*, *stream*> need to be evaluated because they contain the method *wait* in their scopes. Not evaluated are rule instances such as <*rule1*, *play*> or <*rule1*, *select*>.

4.2 Scope Detection and Completeness

Since we treat our consistency rules as non-observable black boxes, the first major obstacle is how to identify the instance-scope of a rule instance even though this scope is not predictable. We do so by instrumenting the modeling tool the engineer uses. We essentially built a profiler for several commercial modeling tools, including IBM Rational Rose. A typical profiler is used during testing to log the lines of code executed. Our UML profiler is similar in that it observes the evaluation of consistency rules and logs the model elements used. Our profiler is based on a COTS-incorporation infrastructure (Section 3.2) which exposes the model elements of a COTS tool, says Rose, through a UML 1.3-compliant interface and provides a change detection mechanism. The detailed working of this profiler cannot be discussed here due to space limitations (see [5]).

We know from Section 4.1 that the model elements that must be in the scope are those model elements accessed during the rule's evaluation. Through the help of the UML profiler, it is simple to detect the scope for any given rule by clearing the log before the evaluation, letting the profiler log all model elements accessed during the evaluation, and storing the accessed model elements.

For example, consider again the evaluation of rule 1 on message wait. This rule starts its evaluation at the root element – a message. It first requests the receiver object for the message by invoking the *getReceiver()* method on the message. The profiler logs the use of the message *wait*. The rule then asks for the base class of the object by invoking *getBase()* on the object *st*. The profiler logs the use of the object *st*. The rule then accesses the operations of the base class (*getBehavioralFeatures()*) and requests the name of each one of them (*getName()*). The class and all its operations are added to the log. The comparison at the end does not access any additional model elements.

The scope of a rule is thus simply the set of model elements accessed during the rule's evaluation. It is computable automatically. The questions are: (1) is this scope complete and (2) is this scope (close to) minimal.

The Scope is Complete

A rule instance's correctness requires its 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 evaluation but not omitting necessary ones.

Our premise is that consistency rules are stateless and deterministic (recall Section 3.1). The same rule invoked on the same model uses the exact same model elements and results in the exact same truth value 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). The profiled scope must thus be complete because it accesses at least the model elements needed for its evaluation. The scope is complete even though a consistency rule may contain AND/OR subconditions that influence how and whether model elements are accessed. For example, rule 1 from Figure 2 could be rewritten in a way that is equivalent in its truth value but uses a slightly different scope:

```
Operations=message.receiver.base.operations
For each operation in operations
    if (operation.name equals message.name) return true
end for
return false
```

This rewritten rule 1 iterates over the set of operations *until* the first operation is found that matches the method name. This expression is equivalent to the logical 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. Thus, if rule 1 evaluates to true then not all accessible model elements are accessed during its evaluation and its scope does not include all accessible elements (e.g., *B*). Clearly this rewritten rule does not have a complete scope if we define a complete scope to include all model elements that are *potentially* accessible.

Fortunately, this level of completeness is not necessary for our problem. A logical condition containing an OR operator must access *at least* those model elements that are required to

[2] The root element of a rule instance is typically the first model element accessed during evaluation. Since every rule accesses at least one model element and the evaluation is deterministic, it follows that *every consistency rule must have a root element*.

determine its truth value. Such a condition cannot change its truth value if a model element is changed that is potentially accessible although it was not accessed. For as long as *A* stays true in *A or B*, changes to *B* do not matter and are not required to be in the scope. For example, <*rule1*, *stream*> evaluates to true because there is a *stream()* operation in the *Streamer* class. The rewritten rule does not access *wait()* because it would only access it after *stream()*. Operation *stream()* is thus in the scope but operation *wait()* is not. This is not a problem because operation *wait()* does not contribute to the truth of this particular rule instance.

We encounter a similar situation with AND subconditions because there a condition must be evaluated only until the first subcondition is false to make the truth value false (e.g., if *A* is false in *A and B* then *B* is not evaluated). Again, not all potentially accessible model elements are accessed if the condition evaluates to false but again this level of completeness is not required.

The scope may change over the life of a rule instance. Fortunately, the scope of a rule instance only changes if a model element in the scope changes. Thus, the re-computation of a rule's scope coincides with the evaluation of its truth value and no additional overhead is required. This also implies that the scope of affected rule instances must be recomputed every time the model changes but this overhead is negligible (i.e., the <50ms evaluation time for 99% of all changes already includes this overhead).

In summary, the scope of a consistency rule cannot be predicted ahead of time. We demonstrated that we can use a profiler to observe it instead - 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. However, the scope does consume memory.

The Scope is Not Minimal but Bounded

A minimal scope guarantees that a rule is evaluated only if its truth value changes. Any evaluation that does not change a 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 rule instance <*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 used change example from *wait* to *suspend*, this change is different in that it does not affect the truth value because there is a corresponding method *stream()*.

It is infeasible to eliminate all unnecessary evaluation without once again introducing manual and error prone change expressions as required for the "what happens if" solution in 3.3. 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. *The instance-based scope is thus bounded to not include model elements that do not potentially affect the truth value.* We evaluated whether this bounded scope is still computationally scalable and Section 6 presents the empirical evidence based on 29 models, tens of thousands of model elements, and over 140,000 rule instances. We found that the scope sizes, while not minimal, were small in including 20 model elements or fewer for 95% of all rule instances. But most significantly, we found that the scope sizes do *not* increase with the size of the model. They are in fact a constant. This explains why 99% of all model changes required 50ms or less evaluation time. Even the worst case was less than 2 seconds and this worst case occurred extremely rarely in only 0.000003% of all model changes.

In summary, our scope is 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 rule types. This is a known scalability issue of consistency checking and discussed in more detail later.

Recognizing Rule Instances

If a model element changes then all those rule instances are affected (i.e., should be evaluated) that contain the changed element in their scopes. A simple lookup table is sufficient to efficiently locate all affected rule instances for any given changed model element:

```
processChange(changedElement)
  for every rule instance where scope contains changedElement
    evaluate <rule, changedElement>
  end for
```

Obviously, a scope is needed to know when to evaluate a rule instance. But given that the scope is only available *after* the first evaluation, how does the first evaluation happen (i.e., the chicken and the egg problem)? The following discusses how to create and first evaluate rule instances; and how to destroy them when they are no longer needed. For example, if an engineer creates the message *connect* between two objects in a sequence diagram (Figure 5) then there are no rule instances yet that could evaluate its truth value.

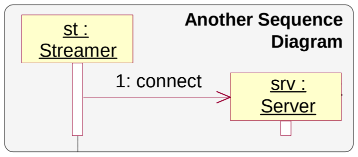


Figure 5. A Message is Created in a Sequence Diagram

We thus require a mechanism for creating rule instances that is based on the *types* of changes that could happen. The most simplistic way of recognizing a new rule instance is through the *required type* of its root element. That is, a new rule instance must be created when a model element is created that matches the type of the rule's root element. For example, we know that rule 1 requires an UML *Message* as its root element because it was written from the perspective of a message (recall Figure 2). Thus, once the message *connect* is created, a new rule instance <*rule1*, *connect*> is created and evaluated also (top of Figure 6).

The change notification mechanism of the UML 1.3 infrastructure distinguishes between the creation, modification, and deletion of a model element. If a model element is created then all rules (such as the ones in Figure 2) that have a type of root element equal to the type of the changed element must be found. For every rule found, a rule instance is created with the changed model element as its root element. For example, after the creation of the message *connect* we find that rules 1, 2, and 3 have the same type of root element. These rules are then instantiated with the message *connect* as the root element (i.e., resulting in three new rule instances). A newly created rule instance is immediately evaluated to compute its truth value and scope. The type of the root element can be determined introspectively and need not be provided in form of additional annotations (Figure 6).

```
processChange(changedElement)
  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
```

Figure 6. Algorithm for Processing a Change Instantly

Rule instances are destroyed once their root elements are deleted. For example the above three rule instances are destroyed once the message *connect* is deleted. Thus, when a model element is deleted, we find all *rule instances* with the same root element as the changed element. These rule instances are then destroyed.

The life of a rule instance is tied to the life of its root element. The root element remains constant for any given rule instance throughout its life. It is interesting to observe that the creation of rule instances is based on type information (types of model elements) whereas the evaluation and destruction of rule instances are based on instance information (model elements). 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 rule instances and the evaluation of others.

4.3 Evaluation Buffering

Our approach initially buffers the evaluation of rule instances. The reason for this is that single model changes typically cause multiple change notifications. For example, if the class type (base) of object *svr* is changed from *Server* to *Streamer* then this results in three change notifications:

Modified model element: 105 UML.Object [base]
Modified model element: 106 UML.Class [objects]
Modified model element: 107 UML.Class [objects]

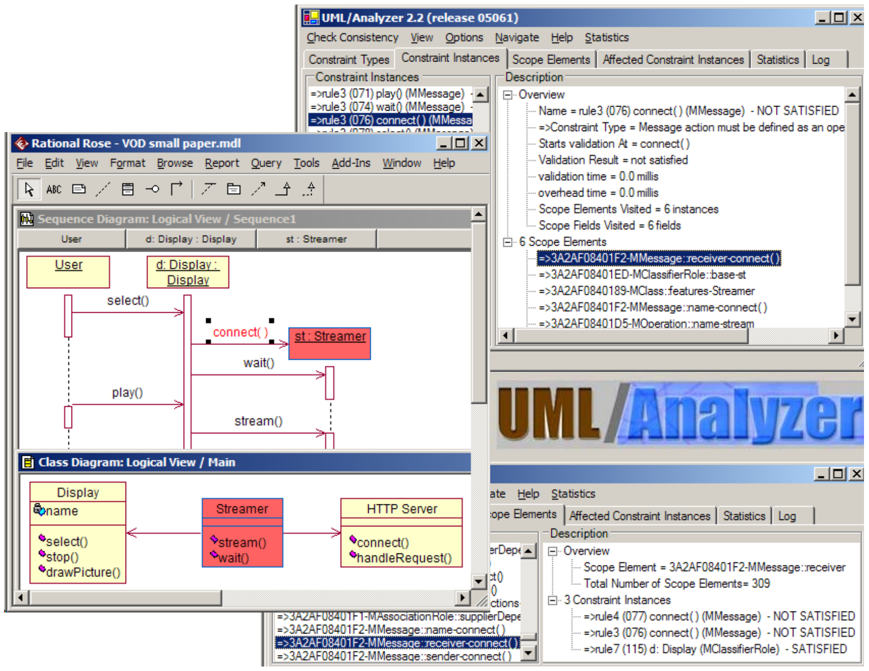
The first change notifies that the *base* field of the object was changed and the second/third ones notify that the *objects* fields of both classes changed (the back pointers) - recall 3.2. If we were to investigate the three change notifications separately then we would duplicate the evaluations of rule instances. For example, two of the three changed model elements are in the scope of <*rule1*, *connect*> but the rule need only be evaluated once. Our approach processes all change notifications first before evaluating any rule instance. The purpose of the evaluation buffer is thus to avoid adding the same rule instance twice. This mechanism also prevents the double evaluation of a newly created rule instance (Figure 6 appears to evaluate new rule instances twice).

In summary, the buffering of evaluation rules improves the response time. The previously reported response time of less than 50ms for 99% of all model changes included this buffering. It must be stressed that this buffering is not a delay mechanism but simply a way of separating the processing of change notification from the evaluation of the affected rule instances. The evaluation buffer can also be used for consistency rule ordering. UML consistency rules are not ordered in their evaluation. However, it is possible to write consistency rules with a particular ordering in mind. It is out of the scope to discuss this issue further.

5. UML/ANALYZER TOOL

The UML/Analyzer tool implements our instant consistency checking approach (Figure 7). It is built on top of the UML 1.3 infrastructure we previously built for IBM Rational Rose to access Rose model elements and to receive instant change notifications when the Rose model changes. 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 6. To include new consistency rules, the tool provides a consistency rule template which requires an evaluation condition and a bit of administrative overhead for registration. The rules are currently written in a programming language based on the API for accessing model elements through our UML 1.3 infrastructure.

Figure 7 depicts a few screen snapshots of the tool. The left depicts IBM Rational Rose. An inconsistency is highlighted. It shows that the message *connect* (in the sequence diagram) does not have a corresponding operation in the receiver's base class. This inconsistency (described in the top right) involves 6 model



=>rule3 (071) play() (MMessage)
=>rule3 (074) wait() (MMessage)
=>rule3 (076) connect( ) (MMessa

Overview
Name = rule3 (076) connect( ) (MMessage) - NOT SATISFIED
=>Constraint Type = Message action must be defined as an ope
Starts validation At = connect( )
Validation Result = not satisfied
validation time = 0.0 millis
overhead time = 0.0 millis
Scope Elements Visited = 6 instances
Scope Fields Visited = 6 fields
6 Scope Elements
=>3A2AF08401F2-MMessage::receiver-connect( )
=>3A2AF08401ED-MClassifierRole::base-st
=>3A2AF0840189-MClass::features-Streamer
=>3A2AF08401F2-MMessage::name-connect( )
=>3A2AF08401D5-MOperation::name-stream

=>3A2AF08401F1-MAssociationRole::supplierDepe
=>3A2AF08401F2-MMessage::name-connect( )
=>3A2AF08401F2-MMessage::receiver-connect( )
=>3A2AF08401F2-MMessage::sender-connect( )

Overview
Scope Element = 3A2AF08401F2-MMessage::receiver
Total Number of Scope Elements= 309
3 Constraint Instances
=>rule4 (077) connect( ) (MMessage) - NOT SATISFIED
=>rule3 (076) connect( ) (MMessage) - NOT SATISFIED
=>rule7 (115) d: Display (MClassifierRole) - SATISFIED

Figure 7. UML/Analyzer Tool Depicting an Inconsistency in IBM Rational Rose™

elements, which are listed there. 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 rule instances that accessed it. The bottom right shows that the message *connect* is actually involved in two inconsistencies. This bi-directional navigation is essential for understanding and resolving inconsistencies.

6. 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 29 UML models (26 of them were third-party models) ranging from small models to very large ones (Table 1). These models were evaluated on 24 types of consistency rules (the three given in Figure 2 and 21 additional ones). In total, over 140,000 rule instances were evaluated. 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 370.000 field values affected (most model elements had multiple fields), we did so automatically. In 97% of all model changes, the response time was less than 10ms; 99% of all rule instances required less than 50ms with an average of 9ms per change and a worst-case of less than 2 seconds.

Table 1. Study Models used for Empirical Evaluation

Size[3]	Model Name	Size[3]	Model Name
3450	ANTS Visualizer	31478	Insurance Fees&Claims
810	Bank Automat	1899	Inventory and Sales
6459	Biter Robocup Client	4083	iTalks
4741	BMS	3366	LCA
125978	Boeing OEP 3.2	544	Microwave Oven
65213	Boeing PCES	891	MVC
6967	Calendarium 2.1	3605	NPI
1409	Curriculum	2321	NZ Intern. Airport
4766	DeSI 2.3	38719	OODT
20554	DSpace 3.2	1729	Teleoperated Robot
1113	eBullition	1209	UML Tutor
4298	Game System	3067	Vacation and Sick Leave
2352	HDCP Defect Seeding	230	Video on Demand
5014	HMS	23016	Wordpad
1596	Home Appliances & Ctrl		

Initial Cost of Computing All Truth Values and Scopes: The cost of a single evaluation of a rule instance is approximately the number of fields visited (=scope size S_{size}). The number of rule instances of a rule type $RT_{\#}$ is at most the number of existing model elements M_{size}. The computational complexity for evaluating all rule instances is thus $O(RT_{\#} * M_{size} * S_{size})$. This cost is a one-time expense.

Recurring Cost of Computing Changed Truth Values and Scopes: For every changed model element, it is necessary to identify all rule instances that are affected. We define the number of affected rule instances as *ARI*. The computational cost for evaluating all affected rule instances is thus $O(ARI * S_{size})$. This cost is a recurring cost because it applies to every model change.

We applied our instant consistency checking tool (the UML/Analyzer) to the 29 sample models and measured the scope sizes S_{size} and the *ARI* by considering all possible model changes[4]. This was done through automated validation by systematically changing all model elements. In the following, we present empirical evidence that S_{size} and *ARI* 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} (min/max) but found that the averages stayed constant with the size of the model. Figure 8 depicts the values for S_{size} relative to the model sizes of the 29 sample models. The figure depicts each model as a vertical range (minimum to maximum). The solid dots between the minimum and maximum values are the average values. Notice the constant, horizontal line of average scope sizes.

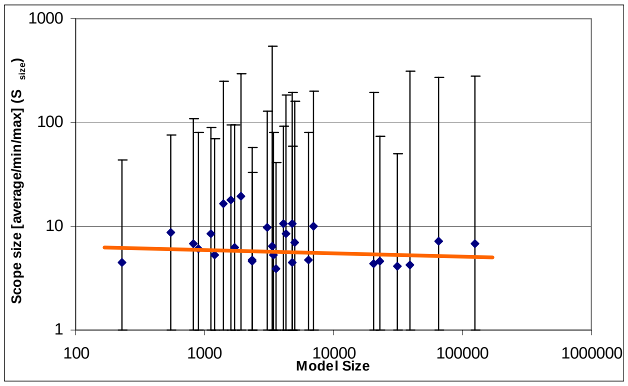


Figure 8. Scope Sizes remain constant over Model Size

The initial, one-time cost of computing the truth values and scopes of a model is linear with the size of the model and the number of rule types $O(RT_{\#} * M_{sizee})$ 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 rule instances must be evaluated with a single change (*ARI*). Since the scope sizes were constant, it was expected that the *ARI* would be constant also (i.e., the likelihood for rule instances to be affected by a change is directly proportional to the scope size). Again, we found a wide range of values between the smallest and largest *ARI* (min/max) but confirmed that the averages stayed constant with the size of the model (Figure 9) – though the maximum increased slightly (consider the logarithmic scale of the x-axis).

ARI was computed by evaluating all rule instances and then measuring in how many scopes each model element appeared. The figure shows that in the worst case, over 1000 rule instances have to be evaluated. But the average values reveal that most

[3] We only counted fields of model elements that were used during consistency checking so that the sizes were not artificially inflated.

[4] We only changed model elements but did not create/delete them because there would be an infinite number of possible model changes. Also, creation/deletion causes changes to the model size only and we will demonstrate that our approach's scalability is not affected by the size.

changes require few evaluations (between 1-10 depending on the model).

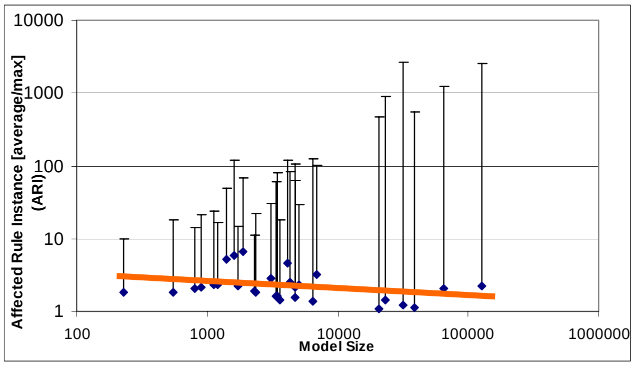


Figure 9. Affected Rule Instances (ARI) stays constant

However, not all changes were equally likely. While we were not allowed to observe how the engineers changed the models, we do know that rule instances with bigger scopes have a higher likelihood of them being re-evaluated. The scalability issue was thus the scope size S_{size}. We showed that S_{size} does not increase with the size of the model which is one important factor for scalability. The following shows the distribution of S_{size} across the 140,000 rule instances of our 29 models. While there were some outliers where S_{size} was large, we found that the likelihood of S_{size} being large decreased drastically.

Figure 10 depicts for all 29 projects separately what percentage of rule instances (y-axis) had a scope of less than 5, 10, 15,… model elements (x-axis). The table shows that over 72% of all rule instances evaluated less than 6 model elements and only 5% of all rule instances evaluated more than 25 model elements. Therefore, 95% of all 140,000 rule instances evaluated less or equal than 25 model elements. This data further confirms that even in the worst-case scenario of over one thousand evaluations per change, most of these evaluations are expected to be very cheap.

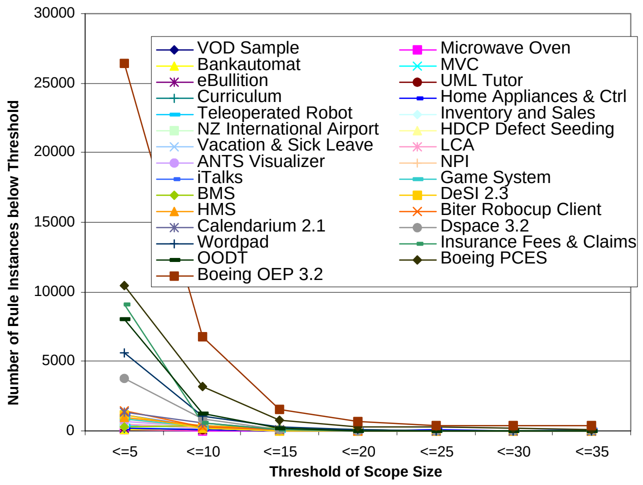


Figure 10. Number of Model Elements Accessed by Constraints

The above data did not consider a changing number of consistency rules $RT_{\#}$. Indeed, consistency rules tend to be well-defined, finite, and stable in most domains. However, as was discussed earlier, we worked with engineers who adapted these rules and even introduced their own. 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.

Our approach even then outperforms a type-based approach if the creation of new model elements is the dominant operation. Recall that our approach uses type information for the creation of rule instances but there can only be as many creations as there are rule types (a constant independent of the size of the model). Thus, our type-based creation does not have the scalability problems of a type-based approach to consistency checking (Figure 3).

On the downside, our approach does require additional memory for storing the scopes. Figure 11 depicts the linear relationship between the model size and the memory cost. It can be seen that the memory cost rises linearly with the number of rule instances. This should not be surprising given that the scope sizes are constant with respect to the model size but the number of rule instances increases.

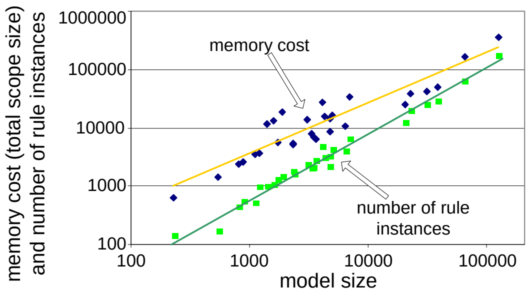


Figure 11. Memory Cost Increases Linearly with Model Size

We found that there were about half as many rule instances as the number of model elements (given our 24 consistency rule types). Thus, there were in average 0.02 rule instances per model element and consistency rule. Given that a rule instance had in average 7.6 scope elements, it followed that the memory cost was 0.15 * model size * number of consistency rules ($RT_{\#}$). Or its cost is O($RT_{\#}$*S_{size}). This linear memory cost was acceptable to the engineers given the vastly superior response time.

Threats to Validity:

Internal validity: We investigated 24 consistency rules in the context of 29, mostly third-party models. The models were vastly different in size and domain. Since our approach performed well for *all* these models, we believe that there are no threats to the internal validity of the measured data. However, we were not able to directly observe the engineers in their use of our approach and could not provide heuristics on likely changes. Yet, in interviews with the engineers we were told that at no time they felt delays of any kind. In their opinion, the approach was truly instant.

External validity: We conducted experiments where we changed the number of constraint rules and found the basic observations to be identical although the values changed. For example, we observed that a different number of consistency rules also produced a constant *ARI* although the *ARI* value was different (i.e., half the number of consistency rules resulted in roughly half the *ARI*). It follows that more consistency rules imply more evaluation time. This cost is expected to increase linearly. *Clearly, we cannot support an infinite number of consistency rules but we typically do not have to*. For the engineers we worked with, the 24 consistency rules covered all relevant situations for the consistency of sequence diagrams with class and statechart diagrams (in their domains). And there are roughly a hundred more known rules for other types of UML diagrams, say deployment diagrams or use-case diagrams. 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 no necessarily imply a longer evaluation time. However, given that 99% of all changes required 50ms or less evaluation time, we do not foresee scalability issues even with an order of magnitude larger $RT_{\#}$.

7. CONCLUSIONS

This paper introduced an approach for quickly, correctly, and automatically deciding when to evaluate consistency rules. We demonstrated that our approach works with black-box consistency rules and that these rules do not have to be annotated. Instead, our approach used a form of profiling to observe the behavior of the consistency rules during evaluation. We demonstrated on 29 UML models that the average model change cost 9ms, 99% of the model changes cost less than 50ms, 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 instance of consistency rule. This scope is computed automatically during evaluation and used to determine when to re-evaluate rules. 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 a part of the model of interest. This is important for living with inconsistencies but it is also important for not getting overwhelmed by 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.

8. REFERENCES

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

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

[3] Cheng, B. H. C., Wang, E. Y., and Bourdeau, R. H.: *A Graphical Environment for Formally Developing Object-Oriented Software*, Proceedings of IEEE International Conference on Tools with AI, November 1994.

[4] Movie Player at http://peace.snu.ac.kr/dhkim/java/MPEG/.

[5] Egyed A. and Balzer B.: Integrating COTS Software into Systems through Instrumentation and Reasoning. *Journal of Automated Software Engineering* (JASE) 13(1), 2006, 41-64.

[6] Emmerich, W.: *GTSL | An Object-Oriented Language for Specication of Syntax Directed Tools*, Proceedings of the 8th International Workshop on Software Speciation and Design, 1996, pp.26-35.

[7] Fickas, S., Feather, M., Kramer, J.: *Proceedings of the Workshop on Living with Inconsistency*. Boston, USA, 1997.

[8] Finkelstein A., Gabbay D., Hunter A., Kramer J., and Nuseibeh B.: Inconsistency Handling in Multi-Perspective Specifications, *IEEE Transactions on Software Engineering* (TSE) 20(8), 1994, 569-578.

[9] Habermann A. N. and Notkin D.: Gandalf: Software Development Environments. *IEEE Transactions on Software Engineering*, 12(12), 1986, 1117.

[10] Hunter A. and Nuseibeh B.: Managing Inconsistent Specifications: Reasoning, Analysis, and Action. *ACM Transactions on Software Engineering and Methodology* 7(4), 1998, 335-367.

[11] Kaplan, S. M. and Kaiser, G. E.: *Incremental attribute evaluation in distributed language-based environments*, Proceedings of the 5th ACM Symposium on Principles of Distributed Computing, Calgary, Canada, 1986, pp.121-130.

[12] Mackworth A. K.: Consistency in Networks of Relations. *Journal of Artificial Intelligence*, 8(1), 1977, 99-118.

[13] Nentwich C., Capra L., Emmerich W., and Finkelstein A.: xlinkit: a consistency checking and smart link generation service. *ACM Transactions on Internet Technology* (TOIT) 2(2), 2002, 151-185.

[14] Nuseibeh, B. and Russo, A.: *On the Consequences of Acting in the Presence of Inconsistency*, Proceedings of the 9th International Workshop on Software Specification & Design, Ise-Shima, Japan, April 1998, pp.156-158 .

[15] Robins, J. and others: "ArgoUML," http://argouml.tigris.org/.

[16] Roussopoulos N.: An Incremental Access Method for ViewCache: Concept, Algorithms, and Cost Analysis. *ACM Transactions on Database Systems* 16(3), 1991, 535-563.

[17] Taylor, R. N., Selby, R. W., Young, M., Belz, F. C., Clarce, L. A., Wileden, J. C., Osterweil, L., and Wolf, A. L.: *Foundations of the Arcadia Environment Architecture*, Proceedings of the 4th ACM SIGSOFT Symposium on Software Development Environments, Irvine, CA, 1998.

[18] Tsiolakis, A. and Ehrig, H.: *Consistency Analysis of UML Class and Sequence Diagrams using Attributed Graph Grammars*, Proceedings of Workshop on Graph Transformation Systems (GRATRA), March 2000, pp.77-86.

[19] Unified Modeling Language (UML) at http://www.omg.org/.

[20] van Der Straeten, R., Mens, T., Simmonds, J., and Jonckers, V.: *Using Description Logic to Maintain Consistency between UML Models*, Proceedings of 6th International Conference on the Unified Modeling Language (UML 2003), October 2003.