Proceedings of the 29th International Conference on Software Engineering (ICSE), Minneapolis, USA, May 2007, pp: ???

UML/Analyzer: A Tool for the Instant Consistency Checking of UML Models

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

Abstract

Large design models contain tens of thousands of model elements. Designers 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 also hard to keep track of known inconsistencies. The UML/Analyzer tool identifies inconsistencies instantly with design changes and it keeps track of all inconsistencies over time. It 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 is integrated with the UML modeling tool IBM Rational Rose™ for broad applicability and usability. It is highly scalable and was evaluated on dozens of design models.

1. Introduction

Instant error feedback of any kind is a fundamental best practice in the software engineering process. Although, there are several tools [6,7] that support the incremental consistency checking of UML design models [8], none of them have been proven to provide design feedback instantly during modeling. This problem exists in part because correctly deciding what consistency rules to evaluate when a model changes is a seemingly impossible task given the close to infinite number of changes and change combinations. Any manual overhead in deciding this is bound to be error prone.

This paper presents the UML/Analyzer tool for the instant consistency checking of UML models. The tool helps designers in detecting and tracking inconsistencies and it does so correctly and quickly with every design change. The tool is fully automated and does not require manual assistance. The tool can be used to provide consistency feedback in an intrusive or non-intrusive manner. This paper presents the tools and its capabilities. The theoretical background was published in ICSE 2006 [3].

1.1 Illustration and Problem

The illustration in Figure 1 depicts two diagrams created with the UML modeling tool IBM Rational Rose™. The given model represents an early design-time snapshot of a real, albeit simplified, video-on-demand (VOD) system [2]. 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.

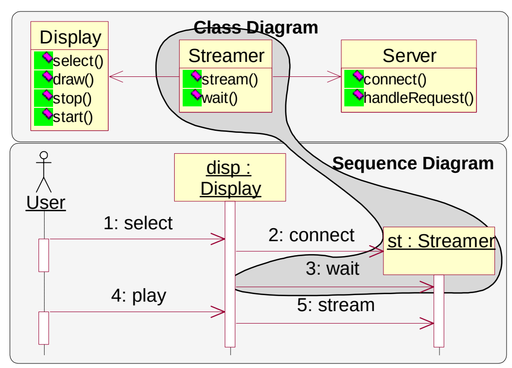

Figure 1. Simplified UML Model of the VOD System

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 any UML model must satisfy for it to be considered a valid UML model. Figure 2 describes two such consistency rules on how UML sequence diagrams (objects and messages) relate to class 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)<>{})

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*, receiver.base is the class *Streamer*, and base.operations is {*stream()*, *wait()*}. 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).

1.2 Detect Inconsistencies

Our tool supports both the batch consistency checking of an entire UML model and the incremental consistency checking of design changes. To support the fast, incremental checking of design changes, the tool identifies all model elements that affect the truth value of any given consistency rule. A consistency rule needs to be re-evaluated if and only if one of these model elements changes. We refer to this set of model elements as the *scope* of a consistency rule. Identifying the scope is simple in principle, however, it is not possible to predict in advance what model elements are accessed by any given consistency rule.

Our tool circumvents this problem by observing the run-time behavior of consistency rules during their evaluation. To this end, we developed the equivalent of a 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 can 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 [5]).

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 earlier). The scope of rule 1 on message *wait* is thus {*wait*, *st*, *Streamer*, *stream()*, *wait()*} as illustrated through the shading in Figure 1. Naturally, this scope is different for every rule and model element it is applies on. 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. The UML/Analyzer tool thus records and maintains the scope separately for every <rule, model element> pair (e.g., <rule1, wait>). We refer to a <rule, model element> pair as a *rule instance*.

If a model element changes then all those rule instances are re-evaluated that include the changed model 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 re-evaluated because they contain the method *wait* in their scopes. Not evaluated are rule instances such as <*rule1*, *play*> or <*rule1*, *select*>.

In earlier work [3], we demonstrated that this scope is complete and correct based on the evaluation of dozens of small to large-scale UML models.

1.3. Track 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 tool provides inconsistencies instantly, it does not require the engineers to fix them instantly. Our tool tracks all presently-known inconsistencies and lets the engineers explore inconsistencies according to their interests in the model.

However, it must be noted that the scope of an inconsistency is continuously affected by model changes. Scopes of inconsistencies must thus be maintained continuously. Fortunately, we found that the scope of a rule instance only then changes if one of the model elements in the scope changes. In other words, the scope of a rule instance changes only if its truth value is affected by a change. So, the mechanism for discovering the scope of a rule instance (discussed earlier in Section 1.2) applies to the tracking of inconsistencies as well. The only difference: our tool re-captures the scope of a rule instance every time the rule is re-evaluated. This way the scope remains up-to-date. The overhead cost of doing so is minimal.

If a designer later on desires to identify all inconsistencies related to a particular model element (or set of model elements) then our tool simply searches through the scopes of all rule instances to identify the ones that are relevant.

1.4. Fixing Inconsistencies

The UML/Analyzer tool also provides support for fixing inconsistencies. It must be noted that in order to fix an inconsistency at least one of the model elements of the scope of that inconsistency must be changed. Thus, the scope of an inconsistency serves as the starting point for fixing inconsistencies. This is a very relevant feature because many existing approaches are unable to pinpoint all the model elements that contributed to any given inconsistency. Our tool provides all this information.

2. Tool and Architecture

Figure 3 depicts a few screen snapshots of the UML/Analyzer 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 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.

Since consistency rules are conditions on a model,

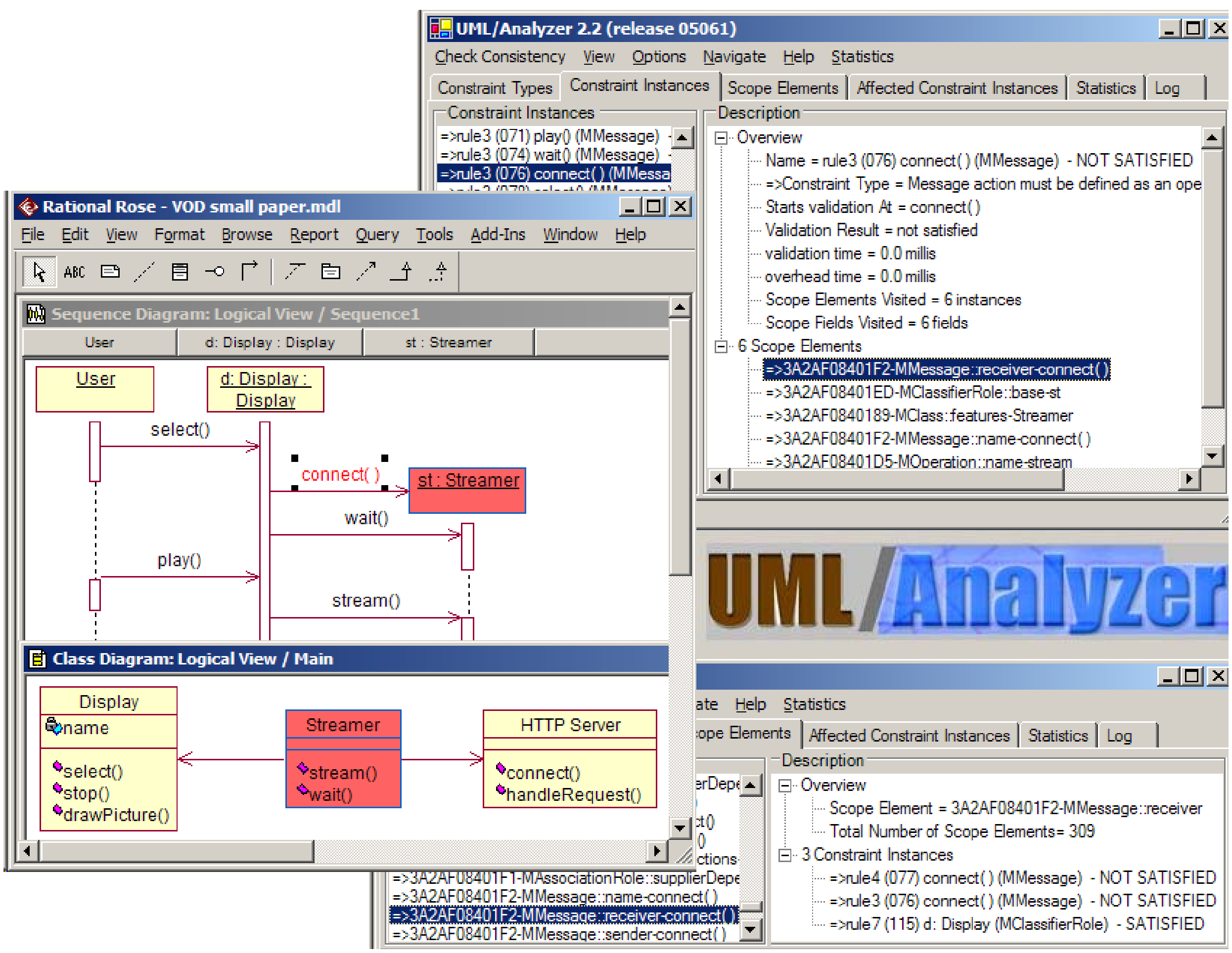

Figure 3. UML/Analyzer Tool Depicting an Inconsistency in IBM Rational Rose™

their truth values change only if the model changes. Instant consistency checking thus requires an understanding when, where, and how the model changes. For this purpose, our tool relies on the UML Interface Wrapper component – an infrastructure we previously developed and integrated with IBM Rational Rose and other COTS modeling tools [4]. This infrastructure exposes the modeling data of the COTS modeling tool in an UML-compliant fashion. It also employs a sophisticated change detection mechanism. The latter is particularly important because it notifies our tool of changes to Rose's UML model.

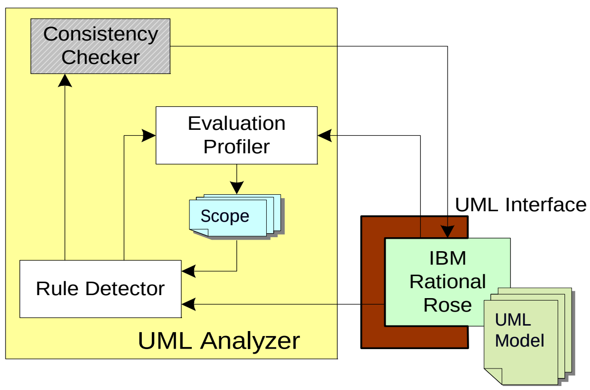

Figure 4. UML/Analyzer Architecture

Figure 4 shows the architecture of our tool. It depicts the modeling tool *IBM Rational Rose* on the lower-right corner. Rose is wrapped by our *UML Interface Wrapper* which provides an UML-compliant API for the *Consistency Checker* (top-left). The *UML Interface Wrapper* also notifies the *Rule Detector* component of user changes to the model. The *Rule Detector* then identifies which consistency rules are affected by the changes. For this purpose, it reads the *Scope* database. The *Rule Detector* then instructs the *Consistency Checker* to re-evaluate the affected consistency rules and it instructs the *Evaluation Profiler* to observe what model elements the *Consistency Checker* accesses. The *Evaluation Profiler* then updates the *Scope* database accordingly.

3. Evaluation

The UML/Analyzer tool has been evaluated on over 40 case studies (industrial and open-source). The tool is not a commercial-grade product; however, it is integrated with the commercial UML modeling tool IBM Rational Rose for ease of use and broader applicability. The tool is part of an ongoing research effort and is continuously evolved and improved upon. As such, there are known bugs and limitations.

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 the 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.

4. References

1. Balzer, R.: "Tolerating Inconsistency," *Proceedings of 13th International Conference on Software Engineering (ICSE-13)*, May 1991, pp.158-165.
2. Dohyung, K.: "Java MPEG Player," *http://peace.snu.ac.kr/dhkim/java/MPEG/*, 1999.
3. Egyed, A.: "Instant Consistency Checking for the UML," *Proceedings of the 28th International Conference on Software Engineering (ICSE)*, Shanghai, China, May 2005.
4. Egyed A. and Balzer B.: Integrating COTS Software into Systems through Instrumentation and Reasoning. *International Journal of Automated Software Engineering (JASE)* 13(1), 2006, pp.41-64.
5. Fickas, S., Feather, M., Kramer, J.: Proceedings of ICSE-97 Workshop on Living with Inconsistency. Boston, USA, 1997.
6. 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, pp.151-185.
7. Robins, J. and others: "ArgoUML," *http://argouml.tigris.org/*.
8. Rumbaugh, J., Jacobson, I., Booch, G.: The Unified Modeling Language Reference Manual. Addison Wesley, 1999.