Developing an Embedded Model for Test suite prioritization process to optimize consistency rules for inconsistencies detection and model changes

Dr. Muzammil H Mohammed[1] and Sultan Aljahdali[2]

[1] Taif University, Saudi Arabia.

Received: 9 December 2011 Accepted: 5 January 2012 Published: 15 January 2012

Abstract

Software form typically contains a lot of contradiction and uniformity checkers help engineers find them. Even if engineers are willing to tolerate inconsistencies, they are better off knowing about their existence to avoid follow-on errors and unnecessary rework. However, current approaches do not detect or track inconsistencies fast enough. This paper presents an automated approach for detecting and tracking inconsistencies in real time (while the model changes). Engineers only need to define consistency rules-in any language-and our approach automatically identifies how model changes affect these consistency rules. It does this by observing the behavior of consistency rules to understand how they affect the model. The approach is quick, correct, scalable, fully automated, and easy to use as it does not require any special skills from the engineers using it. We use this model to define generic prioritization criteria that are applicable to GUI, Web applications and Embedded Model. We evolve the model and use it to develop a unified theory. Within the context of this model, we develop and empirically evaluate several prioritization criteria and apply them to four stand-alone GUI and three Web-based applications, their existing test suites and mainly embedded systems. In this model we only run our data collection and test suite prioritization process on seven programs and their existing test suites. An experiment that would be more readily generalized would include multiple programs of different sizes and from different domains. We may conduct additional empirical studies with larger EDS to address this threat each test case has a uniform cost of running (processor time) monitoring (human time); these assumptions may not hold in practice. Second, we assume that each fault contributes uniformly to the overall cost, which again may not hold in practice.

Index terms—

1 Introduction

here are lots of problems involving the consistency of the software during the development cycle. A lot of cost and investment is put forth to reduce the inconsistency in the software which brings out a consistent software. The main objective of our research is in this area of identifying the inconsistencies in software automatically using various tools and techniques. Also we have hereby focused on the automated model change identification which may also help in identifying the inconsistencies automatically.

Determining the inconsistencies in software automatically will definitely help in reducing the complexity of software maintenance and as well as enhances the performance of the software.

The main focus of the proposed system of automating the consistency checking is on the UML since UML is the basic for any software development.

When we track all the dynamic consistency changes and the rule inconsistencies in the UML we can almost very well say that the software inconsistencies are tracked down, since the software depends on the UML.

In our proposed model of inconsistencies tracking we have laid down the emphasis on the UML rule consistency, UML model changes, Dynamic constraints, meta model constraints, etc.

To identify inconsistencies in an automatable fashion we have devised and applied a view integration framework accompanied by a set of activities and techniques. Our view integration approach exploits the redundancy between views which can be seen as constraints. Our view integration framework enforces such constraints and, thereby, the consistency across views. In addition to constraints and consistency rules, our view integration framework also defines what information can be exchanged and how information can be exchanged. This is critical for scalability and automates ability.

We made use of many tools those analyses the UML and the model to help us in figuring out all the inconsistencies and changes. The major tool is UML analyzer.

(UML/Analyzer is a synthesis and analysis tool to support model-based software development. It implements a generic view integration framework which supports automated model transformation and consistency checking within UML object and class diagrams as well as the C2SADEL architectural description language). If there is a rule flow in your rule project, it reports problems on rules that are included in a rule task, and that may be selected at runtime.

It only compares rules that may be in the same task. In the case of a rule task with dynamic selection filtering, the consistency checking mechanism takes into account the rules that are potentially selected by this task. A rule can be potentially selected when it cannot be established that it definitely cannot be selected.

If there is no rule flow in your rule project, all the rules in the project may be selected.

Consistency checking gives an indication of the consistency of your rules but cannot identify all potential problems. An empty Consistency checking report is therefore not a guarantee that there are no problems in the analyzed rules. b) Rules that are never selected Rules are reported as ”never selected” when they are not part of a rule task and cannot be selected at runtime. For more information, see Rule selection and Rule overriding. c) Rules that never apply This occurs when the conditions of the rule can never be met.

Typically, the syntax of such rules is correct but the rules contain common logic errors. For example:

The wrong operator is used to combine condition statements, for example and instead of or: the category of the customer is Gold and the category of the customer is Platinum.

Values are inverted, for example, in the following rule: the age of the customer is between 70 and 50.

Values in the conditions are not within the permitted range.

2 d) Rules with range violation

In order to reduce the risk of errors, some members can only be assigned values within a specified range. For example, the yearly interest rate on a loan may be limited to values between 0 and 10.

If a rule contains an action that tries to assign a value that is not within the permitted range, Rule Studio displays a range violation error in the report and in the Rule Editor.

3 e) Rules with equivalent conditions

This occurs when two rules contain condition parts that have the same meaning and their actions are different although conflict.

Rules with equivalent conditions do not necessarily represent an error situation, but they may be good candidates to be merged.

4 f) Equivalent rules

Equivalent rules are reported when both their conditions and actions are the same.

In the following example, Rule1 and Rule2 are equivalent:

5 C

Although the syntax of these two rules is different, rule analysis evaluates the numeric expressions and reports that the rules are equivalent. You can therefore delete one of them.

6 2012

7 Year

8 Note

Equivalent rules often arise between a decision table that you create and an existing rule.

9 g) Redundant rules

When two rules have the same actions, one of them becomes redundant when its conditions are included in the conditions of the other.

In the following example, the Else part of Rule2 makes Rule1 redundant: Although Rule1 is correct, it is redundant and can therefore be deleted.

10 Note

Redundant rules often arise between a decision table that you create and an existing rule. h) Conflicting and self-conflicting rules i. Conflicting rules Rules may conflict when the actions of two different rules set a different value for the same business term (member). Conflicts occur in these two rules in circumstances in which the conditions are equivalent or cover the same values.

11 Rule 1 if

the loan report is approved and the amount of the loan is at least 300 000 then set the category of the borrower to Gold Rule 2 if the age of the latest bankruptcy of the borrower is less than 1 and the category of the borrower is not Platinum then set the category of the borrower to No Category Rule1 and Rule2 will conflict when the loan report is approved, the amount of the loan is 300000 (or more), the borrower has not had a bankruptcy in the last year, and the category is anything but Platinum. In these specific circumstances, the rules will set the category of the borrower to different values.

Conflicting rules can be corrected by changing the conditions, deleting one of the rules, or setting different priorities on the rules.

12 ii. Self-conflicting rules

A rule is self-conflicting when two executions of a rule assign different values to the same member. For example, a self-conflicting rule: may apply twice on a given working memory (and ruleset parameters) will set different values to a common attribute For example: if the customer category is Gold then set the discount of the cart to the bonus points of the customer If there are two customer objects with different bonus points in the working memory, the rule is executed twice and a conflict occurs because the two executions of the rule set different values to the discount of the cart.

13 i) Decision table conflicts

To check decision tables, you need to enable the option Include decision tables and decision trees in the inter-rule checks.

This The UML/Analyzer tool, integrated with IBM Rational Rose&8482;, 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-theart. To identify inconsistencies in an automatable fashion we have devised and applied a view integration framework accompanied by a set of activities and techniques. Our view integration approach exploits the redundancy between views which can be seen as constraints. Our view integration framework enforces such constraints and, thereby, the consistency across views. In addition to constraints and consistency rules, our view integration framework also defines what information can be exchanged and how information can be exchanged. This is critical for scalability and automate ability. When the user invokes play, object disp invokes stream on object st. These UML consistency rules describe conditions that a UML model must satisfy for it to be considered a valid UML model. Fig. **??** lists 24 such rules covering consistency, well-formedness, and best practice criteria among UML class, sequence, and statechart diagrams. The first four consistency rules are elaborated on for better understanding. Note that these consistency rules apply to UML only. For the other modeling notations, different consistency rules were needed, which are not described here.

14 Figure 4 : Class Diagram

A consistency rule may be thought of as a condition that evaluates a portion of a model to a truth value (true or false). For example, consistency rule 1 states that the name of a message must match an operation in the receiver's class.

If this rule is evaluated on the third message in the sequence diagram (the wait message), then the condition first computes operations ¼ 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.

IV.

15 Implementation a) Inconsistencies

We use the term inconsistency to denote any situation in which a set of descriptions does not obey some relationship that should hold between them. The relationship between descriptions can be expressed as a consistency rule against which the descriptions can be checked. In current practice, some rules may be captured in descriptions of the development process; others may be embedded in development tools. However, the majority of such rules are not captured anywhere.

Here are three examples of consistency rules expressed in English: 1. In a dataflow diagram, if a process is decomposed in a separate diagram, the input flows to the parent process must be the same as the input flows to the child data flow diagram. 2. For a particular library system, the concept of an operations document states that user and borrower are synonyms. Hence, the list of user actions described in the help manuals must correspond to the list of borrower actions in the requirements specification.

16 Coding should not begin until the Systems

Requirement Specification has been signed off by the project review board. Hence, the program code repository should be empty until the status of the SRS is changed to "approved."

Figure **??** : Manage Inconsistency In our framework, when you iterate through the consistency management process, you expand and refine the set of consistency rules. You will never obtain a complete set of rules covering all possible consistency relationships in a large project. However, the rule base acts as a repository for recording those rules that are known or discovered so that they can be tracked appropriately.

Consistency rules can emerge from several sources:

? Notation dentitions. Many notations have welldefined syntactic integrity rules. For example, in a strongly typed programming language, the notation requires that the use of each variable be consistent with its declaration. ? Application domains. Many consistency rules arise from domain-specific constraints.

17 b) Monitoring and diagnosing inconsistency

With an explicit set of consistency rules, monitoring can be automatic and unobtrusive. If certain rules have a high computational overhead for checking, the monitoring need not be continuous-the descriptions can be checked at specific points during development, using a lazy consistency strategy.

Our approach defines a scope for each rule, so that each edit action need be checked only against those rules that include in their scope the locus of the edit action.

When you find an inconsistency, the diagnosis process begins. Diagnosis includes parts of a description have broken a consistency rule;

? identifying the cause of an inconsistency, normally by tracing back from the manifestation to the cause; and

? classifying an inconsistency.

Classification is an especially important stage in the process of selecting a suitable handling strategy.

Inconsistencies can be classified along a number of different dimensions, including the type of rule broken, the type of action that caused the inconsistency, and the impact of the inconsistency.

18 c) Handling inconsistency

The choice of an inconsistency-handling strategy depends on the context and the impact it has on other aspects of the development process. Resolving the inconsistency may be as simple as adding or deleting information from a software description. But it often relies on resolving fundamental conflicts or making important design decisions. In such cases, immediate resolution is not the best option. You can ignore, defer, circumvent, or ameliorate the inconsistency.

Sometimes the effort to fix an inconsistency is significantly greater than the risk that the inconsistency will have any adverse consequences. In such cases, you may choose to ignore the inconsistency. Good practice dictates that such decisions should be revisited as a project progresses or as a system evolves.

Deferring the decision until later may provide you with more time to elicit further information to facilitate resolution or to render the inconsistency unimportant. In such cases, flagging the affected parts of the descriptions is important.

Sometimes software developers won't regard a reported inconsistency as an inconsistency. This may be because the rule is incorrect or because the inconsistency represents an exception to the rule. In these cases, the inconsistency can be circumvented by modifying the rule or by disabling it for a specific context.

Sometimes, it may be more cost-effective to ameliorate an inconsistency by taking some steps toward a resolution without actually resolving it.

This approach may include adding information to the description that alleviates some adverse effects of an inconsistency and resolves other inconsistencies as a side effect.

19 d) Measuring inconsistency

For several reasons, measurement is central to effective inconsistency management. Developers often need to know the number and severity of inconsistencies in their descriptions, and how various changes that they make affect these measures. Developers may also use given a choice, which is preferred.

Sometimes developers need to prioritize inconsistencies in different ways to identify inconsistencies that need urgent attention. They may also need to assess their progress by measuring their conformance to some predefined development standard or process model.

The actions taken to handle inconsistency often depend on an assessment of the impact these actions have on the development project. Measuring the impact of inconsistency-handling actions is therefore a key to effective action in the presence of inconsistency. You also need to assess the risks involved in either leaving an inconsistency or handling it in a particular way.

The 24 rules were chosen to cover the needs of our industrial partners. They cover a significant set of rules and we demonstrated that they were handled extremely efficiently. But it is theoretically possible to write consistency rules in a no scalable fashion. constraints -that is constraints that may be added, removed, or modified at will without losing the ability for instant, incremental consistency checking and without requiring any additional, manual annotations.

20 Such dynamic. Table 1 : Rules and Description

Constraints arise naturally in many domain specific contexts In addition to meta model constraints, this work also covers application specific model constraints that are written from the perspective of a concrete model at hand (rather than the more generic meta model). We will demonstrate that model constraints can be directly embedded in the model and still be instantly and incrementally evaluated together with meta model constraints based on the same mechanism. For dynamic constraints, any constraint language should be usable. We demonstrate that our approach is usable with traditional kinds of constraint languages (e.g., OCL [5]) and even standard programming languages (Java or C#). Furthermore, our approach is independent of the modeling language used. We implemented our approach for UML 1. element of constraint C1 in Fig. 3 is a UML Message (a meta model element). This implies that this constraint must be evaluated for every instance of a Message in a given model. In Fig. 3 there are three such messages. Model constraints, on the other hand, are written from the perspective of a model element (an instance of a meta model element). Hence, its context element is a model element.

Fig. 6 shows that for every meta model constraint a number of constraint instances are instantiated (top right) -one for each instance of the meta model element the context element refers to. On the other hand, a model constraint is instantiated exactly once -for the model element it defines.

21 Constraint Instance = <constraint, model element >

While the context elements differ for model and meta model constraints, their instances are alike: the instances of meta model constraints and the instances of model constraints have model elements as their context element. The only difference is that a meta model constraint results in many instances whereas a model constraint results in exactly one instance. Since the instances of both kinds of constraints are alike, our approach treats them in the same manner. Consequently, the core of our approach, the model profiler with its scope elements and reevaluation mechanism discussed above, functions identical for both meta model constraints and model constraints as is illustrated in Fig. 6. The only difference is in how constraints must be instantiated. As discussed above, we support the definition of both meta model and model constraints in Java, C#, and OCL. These languages are vastly different but our approach is oblivious of these differences because it cares only about a constraint's evaluation behavior and not its definition. The key to our approach is thus in the model profiling which happens during the evaluation of a constraint. During the evaluation, a constraint accesses model elements (and their fields). Next, we discuss the algorithm for handling model changes analogous to the discussion above. Thereafter, we discuss the algorithm for handling constraint changes which is orthogonal but similar in structure.

22 g) Model Change

If the model changes then all affected constraint instances must be re-evaluated. Above we discussed that our approach identifies all affected constraint instances through their scopes, which are determined through the model profiler. In addition to the model profiler, we also require a change notification mechanism to know when the model changes. Specifically, we are interested in the creation, deletion, and modification of model elements which are handled differently. Fig. 7 presents an adapted version of the algorithm for processing model changes published in [10]. 1) for meta model constraints, one constraint is instantiated for every model element whose type is equal to the type of the constraint's context element. For example, if the meta model constraint C1 is created a new (Fig. 3) then it is instantiated three times -once for each message in Fig. 3 (<C1, getDevices>, <C1, press>, <C1, turnOn>) because C1 applies to UML messages as defined in its context element. 2) for model constraints, exactly one constraint is instantiated for the model element of the constraint's context element. For example, if the model constraint C4 is defined anew (Fig. 3) then it is instantiated once for the WorkroomThermostat as defined in Fig. **??** (<C4, workroomThermostat>) because this constraint specifically refers to this model element in its context. Once instantiated, the constraints are evaluated immediately to determine their truth values and scopes. If a constraint is deleted then all its instances are destroyed. If a constraint is modified all its constraints are re-evaluated assuming the context element stays the same. If the context element is changed or the constraint is changed from a meta model to a model constraint or vice versa, then the change is treated as the deletion and re-creation of a constraint (rather than its modification).

23 processConstraintChange(changedDefinition)

if changedDefinition was created for every modelElement of type/instance changedDefinition.contextElement constraint = new <changedDefinition, modelElement> evaluate constraint else if changedDefinition was deleted for every constraint of changedDefinition, destroy constraint else if condition of changedDefinition was modified for every constraint of changedDefinition, evaluate constraint else for every constraint of changedDefinition, destroy constraint for every modelElement of type/instance changedDefinition.contextElement constraint = new <changedDefinition, modelElement> evaluate constraint

24 Test Results

25 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.

We expected some variability in Ssize 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 Ssize (average/max), but found that the averages stayed constant with the size of the model. 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 OðRT + M size Þ because Ssize is a small constant and constants are ignored for computational complexity.

To validate the recurring computational cost of computing changed truth values and scopes, we next discuss how many CRIs must be evaluated with a single change (ACRI). Since the scope sizes were constant, it was expected that the ACRI would be constant also (i.e., the likelihood for CRIs to be affected by a change is directly proportional to the scope size). Again, we found a wide range of values for ACRI across the many diverse models but confirmed that the averages stayed constant with the size of the model. Fig. 10 depicts the average ACRI through solid dots and their98 percent maximums.

ACRI was computed by evaluating all CRIs and then measuring in how many scopes each model element appeared. The figure shows that in some cases, many CRIs had to be evaluated (hundreds and more). But the average values reveal that most changes required few evaluations (between 3 and 11 depending on the model). We see that a change t o the association field of an AssociationEnd was the most expensive kind of change, with over 4 ms reevaluation cost, on average. A message name change (as was used several times in this paper) was comparatively cheap, with 0.12 ms to reevaluate, on average. First and foremost, we note that all types of model changes are quite reasonable to reevaluate. This implies that irrespective of how often certain types of changes happen, our approach performs. 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 Previously, we mentioned that most changes required very little reevaluation time and that there were very rare outliers (0.00011 percent of changes with evaluation time >100 ms). The reason for this is obvious in Fig. 12, where we see that it is exponentially unlikely for CRIs to have larger scope sizes (Fig. 12a) or for changes to affect many CRIs (Fig. 12b). We show this datum to exemplify how similar the 34 models are in that regard, even though these models are vastly different in size, complexity, and domain.

26 Year

The table shows that over 95 percent of all CRIs accessed less than 15 fields of model elements (scope elements). Fig. 12b depicts for all 34 models separately what percentage of changes (yaxis) affected <¼ 2; 4; 6; . . . CRIs. The table shows that 95 percent of all changes affected fewer than 10 CRIs (ACRI).

The data thus far considered a constant number of consistency rules (24 consistency rules). However, the number of consistency rules is variable and may change from model to model or domain to domain. Clearly, our approach (or any approach to incremental consistency checking) is not amendable to arbitrary consistency rules. If a rule must investigate all model elements, then such a rule's scope is bound to increase with the size of the model. However, we demonstrated on the 24 consistency rules that Rules typically are not global; they are, in fact, surprisingly local in their investigations. This is demonstrated in Fig. 13, which depicts the cost of evaluating changes for each consistency rule separately. Still, each consistency rule takes time to evaluate and Fig. 13 is thus an indication of the increase in evaluation cost in response to adding new consistency rules.

We see that the 24 consistency rules took, on average, 0.004-0.21 ms to evaluate with model changes. Each new consistency rule thus increases the evaluation time of a change by this time (assuming that new consistency rules are similar to the 24 kinds of rules we evaluated). The evaluation time thus increases linearly with the number of consistency rules (RT#).

It is important to note that the evaluation was based on consistency rules implemented in C#. Rules implemented in Java were slightly slower to evaluate but rules implemented in OCL **??**38] were comparatively expensive due to the high cost of interpreting them. On the downside, our approach does require additional memory for storing the scopes. Fig. 14 depicts the linear relationship between the model size and this memory cost. It can be seen that the memory cost rises linearly. This should not be surprising given that the scope sizes are constant with respect to the model size but the number of CRIs increases linearly. As with the evaluation time, this cost also increases with the number of consistency rules (RT#). The memory cost is thus RT# + Ssize . For scalability, this implies a quite reasonable trade-off between the extensive performance gains over a linear (and thus scalable) memory cost. To put this rather abstract finding into a practical perspective, the scope is maintained as a simple hash table referencing the impacted CRIs in form of arrays. With the largest model having over 400,000 scope elements, each of which affects fewer than 10 CRIs, the memory cost is thus equivalent to 400,000 arrays of fewer than 10 CRIs each-quite manageable with today's computing resources. The memory cost stays the same if the scope is stored persistently, in which case the recomputation of the scope upon model load is no longer required.

27 ii. Usability

One key advantage of our approach is that engineers are not limited by the modeling language or consistency rule language. We demonstrated this by implementing our approach on UML 1.3, UML 2.1, Matlab/Stateflow, and Dopler Product Line, and using a wide range of languages to describe consistency rules (from Java, C# to the interpreted OCL). But, most significantly, engineers do not have to understand our approach or provide any form of manual annotations (in addition to writing the consistency rule) to use it. These freedoms are all important for usability. This paper does not address how to best visualize inconsistencies graphically. Much of this problem has to do with human-computer interaction and future work will study this. This paper also does not address downstream economic benefits: For example, how does quicker (instant) detection of inconsistencies really benefit software engineering at large. How many p roblems are avoided, how much less does it cost to fix an error early on as compared to later on? These complex issues have yet to be investigated.

However, as an anecdotal reference, it is worth pointing out that nearly all programming environments today support instant compilation (and thus syntax and semantic checking), which clearly benefits programmers. We see no reason why these benefits would not apply to modeling.

28 VI.

29 Conclusion

The main issues addressed in this paper includes -identifying the inconsistencies correctly and quickly in an automated fashion by reducing the complexity, cost and the effort Next, to evaluate the consistency rules which are not necessarily to be written in special language and special annotations our approach used a form of profiling to observe the behavior of the consistency rules during evaluation. We demonstrated on 34 large-scale models that the average model change cost 1.4 ms, 98 percent of the model changes cost less than 7 ms, and that the worst case was below 2 seconds. It is very significant to understand that our approach maintains a separate scope of model elements for every application (instance) of a consistency rule. This scope is computed automatically during evaluation and used to determine when to reevaluate the rule. In the case of an inconsistency, this scope tells the engineer all of the model elements that were involved. Moreover, if an engineer should choose to ignore an inconsistency (i.e., not resolve it right away), an engineer may use the scopes to quickly locate all inconsistencies that directly relate to any part of the model of interest. This is important for living with inconsistencies but it is also important for not getting overwhelmed with too much feedback at once. This paper significantly identifies the dynamic model changes and a wide variety of consistency rules and the proposals were made for automatic detection and tracking of those inconsistencies and model changes that are static as well as dynamic considering also the cost and the efficiency factors of the automated system that is to be inbuilt as an embedded system to perform the task of automatic detection and embarking techniques to solve the inconsistencies and the model changes in any software development process by using the UML diagram as the base and UML analyzer for evaluation of the constraints and the results are then processed for further actions.

30 VII.

31 Future Work

We cannot guarantee that all consistency rules can be evaluated instantly. The 24 rules of our study were chosen to cover the needs of our industrial partners. They cover a significant set of rules and we demonstrated that they were handled extremely efficiently. But it is theoretically possible to write consistency rules in a nonscalable fashion, although it must be stressed that of the hundreds of rules known to us, none fall into this category. It is future work to discuss how to best present inconsistency feedback visually to the engineer. Also, the efficiency of our approach depends, in part, on how consistency rules are written. [1]

Figure 1:

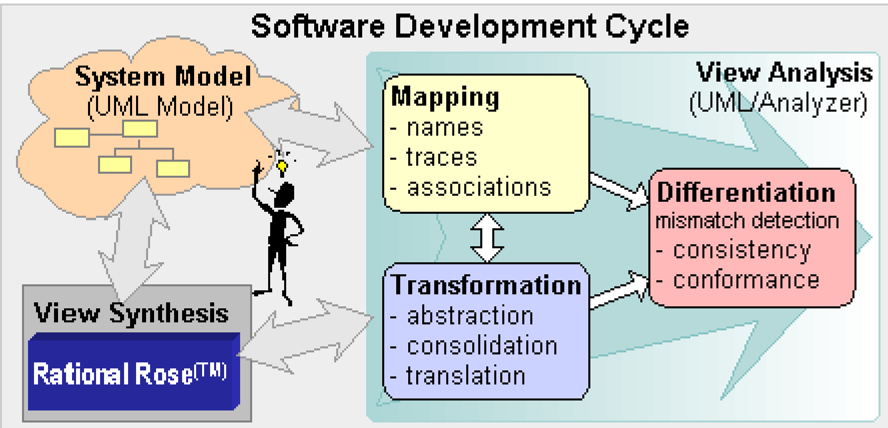
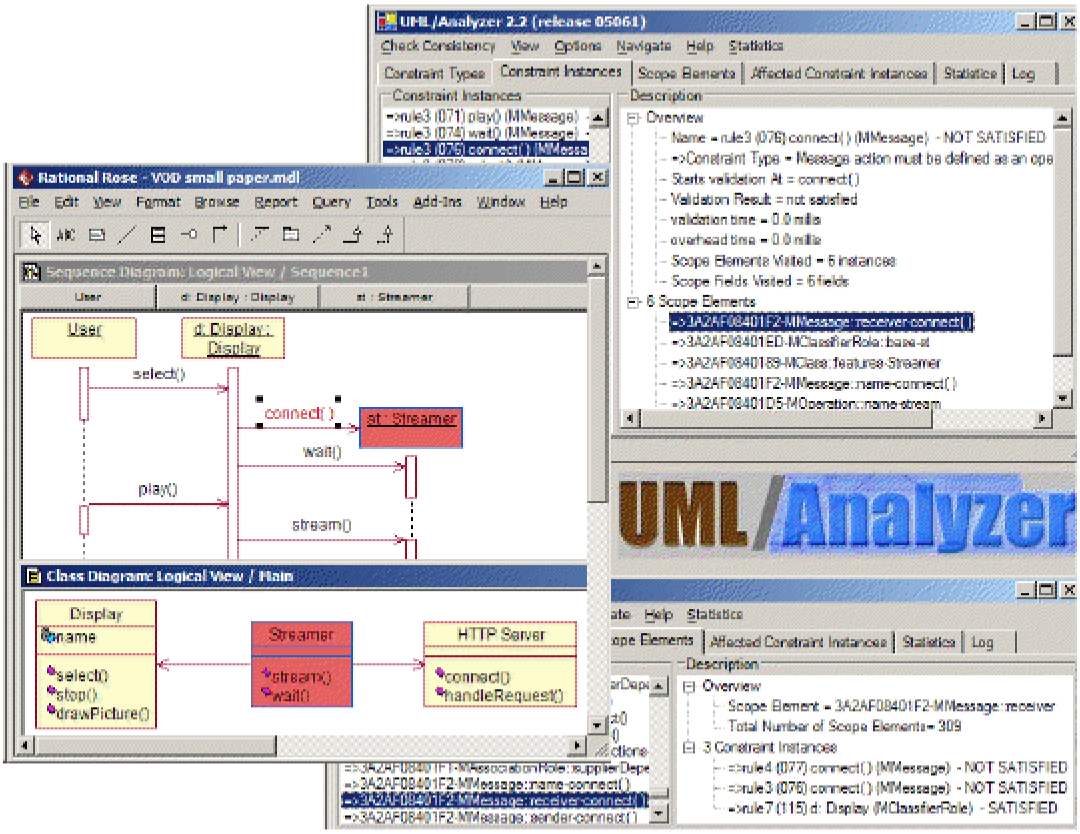
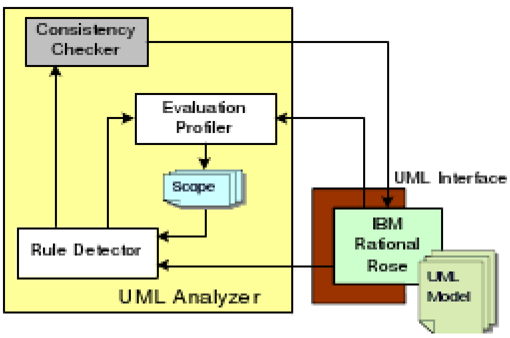


Figure 2:



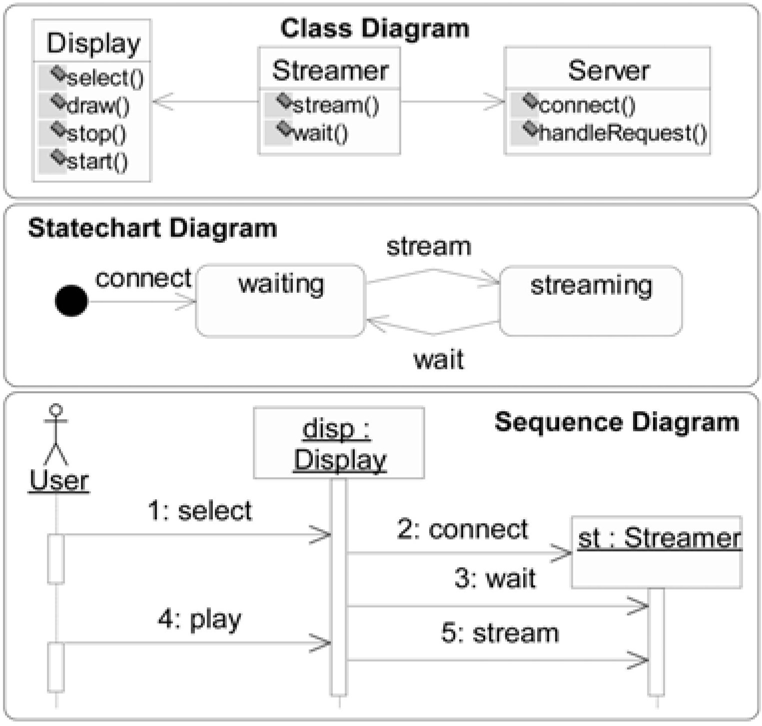
1

Figure 3: Figure 1 :



22

Figure 4: Figure 2 : 2)



3

Figure 5: Figure 3 :

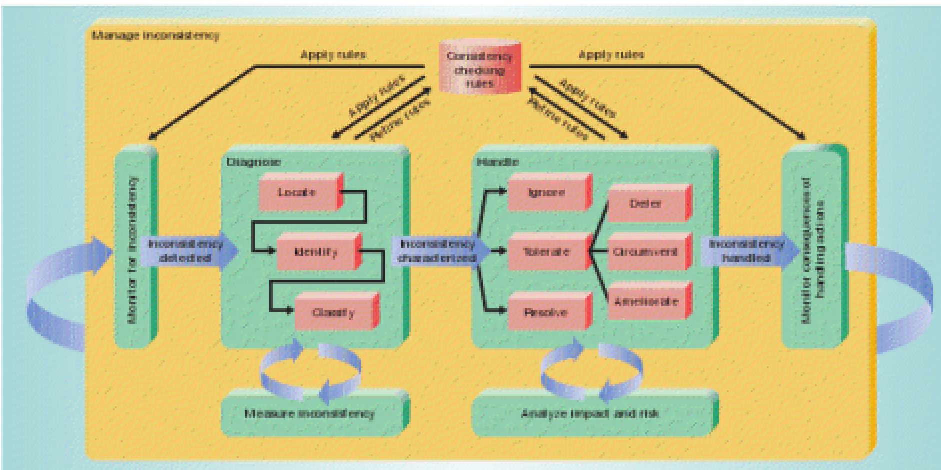


Figure 6:

6

Rule	Description and Implementation
	Name of message must match an operation in receiver's class

Figure 7: Fig. 6

6

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

Figure 8: Figure 6 :

7

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;

Figure 9: Figure 7 :

	return in.intersectsWith(out)
	Sequence of object messages must correspond to events

Figure 10:

8

3	startingPoints = find state transitions equal first message name startingPoints->exists(message sequence equal transition sequence reachable from startingPoint)
4	**Cardinality of association must match sequence interaction**

Figure 11: Figure 8 :

4	**Cardinality of association must match sequence interaction**
5	**Statechart action must be defined as an operation in owner's class**
6	**Parent class attribute should not refer to child class**

Figure 12:

9

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**

Figure 13: Fig. 9 :

1011

9	**At most one association end may be an aggregation or composition**
10	**The connected classifiers of the association end should be included in the namespace of the association**
11	**The class of an association end cannot be an interface if there is an association navigable away from that end**

Figure 14: Fig. 10 :Fig. 11 :

11	**The class of an association end cannot be an interface if there is an association navigable away from that end**
12	**A classifier may not belong by composition to more than one composite classifier**
13	**Method parameters must have unique names**

Figure 15:

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**

Figure 16:

12

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**

Figure 17: Fig. 12 .

13

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**

Figure 18: Fig. 13 :

[Cheng et al. ()] 'A Graphical Environment for Formally Developing Object-Oriented Software'. B H C Cheng , E Y Wang , R H Bourdeau . *Proc. Sixth Int'l Conf. Tools with Artificial Intelligence*, (Sixth Int'l Conf. Tools with Artificial Intelligence) 1994. p. .

[Lee et al. ()] 'Algorithmic Analysis of the Impacts of Changes to Object-Oriented Software'. M Lee , A J Offutt , R T Alexander . *Proc. 34th Int'l Conf. Technology of Object-Oriented Languages and Systems*, (34th Int'l Conf. Technology of Object-Oriented Languages and Systems) 2000. p. .

[Egyed ()] 'Automated Abstraction of Class Diagrams'. A Egyed . *ACM Trans. Software Eng. And Methodology* 2002. 11 p. .

[Campbell et al. ()] 'Automatically Detecting and Visualising Errors in UML Diagrams'. L A Campbell , B H C Cheng , W E Mcumber , K Stirewalt . *Requirements Eng. J* 2002. 7 p. .

[Mackworth ()] 'Consistency in Networks of Relations'. A K Mackworth . *J. Artificial Intelligence* 1977. 8 p. .

[Nentwich et al. ()] 'Consistency Management with Repair Actions'. C Nentwich , W Emmerich , A Finkelstein . *Proc. 25th Int'l Conf. Software Eng*, (25th Int'l Conf. Software Eng) 2003. p. .

[Blanc et al. ()] 'Detecting Model Inconsistency through Operation-Based Model Construc-tion'. X Blanc , I Mounier , A Mougenot , T Mens . *Proc. 30th Int'l Conf. Software Eng*, (30th Int'l Conf. Software Eng) 2008. p. .

[Dhungana et al.] 'DOPLER: An Adaptable Tool Suite for Product Line Engineering'. D Dhungana , R Rabiser , P Gru¨ Nbacher , K Lehner , C Federspiel . *Proc. 11th*, (11th)

[Fickas et al. ()] S Fickas , M Feather , J Kramer . *Proc. ICSE-97 Workshop Living with Inconsistency*, (ICSE-97 Workshop Living with Inconsistency) 1997.

[Egyed ()] 'Fixing Inconsistencies in UML Design Models'. A Egyed . *Proc. 29th Int'l Conf. Software Eng*, (29th Int'l Conf. Software Eng) 2007. p. .

[Habermann and Notkin (1986)] 'Gandalf: Software Development Environments'. A N Habermann , D Notkin . *IEEE Trans. Software Eng* Dec. 1986. 12 (12) p. .

[Egyed et al. ()] 'Generating and Evaluating Choices for Fixing Inconsistencies in UML Design Models'. A Egyed , E Letier , A Finkelstein , ; W Emmerich . *Proc. Eighth Int'l Workshop Software Specification and Design*, (Eighth Int'l Workshop Software Specification and Design) 2008. 1996. 16 p. . (Proc. 23rd Int'l Conf. Automated Software Eng.)

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

[Acar et al. ()] 'Imperative Self-Adjusting Computation'. U A Acar , A Ahmed , M Blume . *Proc. 35th ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages*, (35th ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages) 2008. p. .

[Finkelstein et al. ()] 'Inconsistency Handling in Multi-Perspective Specifications'. A Finkelstein , D Gabbay , A Hunter , J Kramer , B Nuseibeh . *IEEE Trans. Software Eng* 1994. 20 p. .

[Grundy et al. (1998)] 'Inconsistency Manage-ment for Multiple-View Software Development Environments'. J Grundy , J Hosking , R Mugridge . *IEEE Trans. Software Eng* Nov. 1998. 24 (11) p. .

[Kaplan and Kaiser ()] 'Incremental Attribute Evaluation in Distributed Language-Based Environments'. S M Kaplan , G E Kaiser . *Proc. Fifth Ann. Symp. Principles of Distributed Computing*, (Fifth Ann. Symp. Principles of Distributed Computing) 1986. p. .

[Egyed ()] 'Instant Consistency Checking for the UML'. A Egyed . *Proc. 28 th Int'l Conf. Software Eng*, (28 th Int'l Conf. Software Eng) 2006. p. .

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

[Int'l Software Product Line Conf ()] *Int'l Software Product Line Conf*, 2007. p. .

[Egyed and Balzer ()] 'Integrating COTS Software into Systems through Instrumentation and Reasoning'. A Egyed , B Balzer . *Int'l J. Automated Software Eng* 2006. 13 p. .

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

[Lindvall and Sandahl ()] 'Practical Implications of Trace-ability'. M Lindvall , K Sandahl . *J. Software-Practice and Experience* 1996. 26 p. .

[Forgy ()] 'Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem'. C Forgy . *Artificial Intelligence* 1982. 19 p. .

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

[Balzer ()] 'Tolerating Inconsistency'. R Balzer . *Proc. 13 th Int'l Conf. Software Eng*, (13 th Int'l Conf. Software Eng) 1991. p. .

[Easterbrook and Nuseibeh ()] 'Using ViewPoints for Incon-sistency Management'. S Easterbrook , B Nuseibeh . *IEE Software Eng. J* 1995. 11 p. .

[Nentwich et al. ()] 'xlinkit: A Consistency Checking and Smart Link Generation Service'. C Nentwich , L Capra , W Emmerich , A Finkelstein . *ACM Trans. Internet Technology* 2002. 2 p. .