# RULE-BASED DETECTION OF INCONSISTENCY IN UML MODELS

WenQian Liu, Steve Easterbrook and John Mylopoulos

Department of Computer Science, University of Toronto,
Toronto, ON M5S 3G4, Canada.
{wl,sme,jm}@cs.toronto.edu

**Abstract.** Software design inconsistency can be hard to trace manually. Computer assistance in detecting and resolving inconsistency issues can help improve the quality of sophisticated software designs. This paper describes a rule-based (or production system) solution to the aforementioned problem. We characterize classes of inconsistency that occur in software design. We define a production system language and rules specific to software designs modeled in UML. Using this approach, we are able to detect inconsistencies, notify the users, recommend resolutions, and automatically fix the inconsistency during the design process.

## 1 Introduction

Maintaining consistency in a large, evolving design model is difficult. Changes to the model may introduce inconsistencies, which then need to be detected and resolved. A design model is *inconsistent* if it contains conflicting information about the system, and/or violates predefined constraints. Such constraints include both good practices for this kind of design and specific requirements from the stakeholders for this system. The term *inconsistency* refers to a single instance of such conflict or violation. To manually identify and resolve design inconsistencies can be tedious and error prone. Computer assistance in handling design consistency issues is inevitably required.

Recently, a number of research teams have made progress in providing computer-based design consistency checking, notably the *xlinkit* tool [17], Argo/UML [23], and the process-oriented design guidance approach [5]. Inconsistency has also been recognized as a major challenge in requirements engineering [20], and van Lamsweerde has developed a typology of inconsistencies that occur in goal-based requirements models [29]. However, as yet, no such typology has been developed for inconsistency in *design* models. For UML, constraints expressed in OCL [21] are intended to maintain the well-formedness of the UML semantics. We are interested in a broader class of inconsistencies, including problems related to information redundancy, nonconformance to standards and requirements, and the propagation of change through a model as it evolves.

Our goal is to develop a software design environment that automates the detection and resolution of design inconsistencies in design models. Observations of software developers suggest that designers often tolerate inconsistencies in their models because the alternatives may be to leave information out, or to make premature design decisions [19]. Hence, a support environment should not try to prevent inconsistency, and needs

to be flexible in the manner in which it indicates the presence of inconsistencies to the designer. Our approach is to detect and track inconsistencies automatically as the design model is edited, and to suggest likely resolutions where possible. Our approach uses a production system running in the background of an editor. The production rules recognize inconsistencies, and track them by adding entries to the working memory of the production system. The use of a production system gives us a powerful pattern matching approach to inconsistency detection, and allows rules to modify the behavior of other rules, depending on the context.

We define a classification scheme of design inconsistencies that occur in the design representation, and develop our production rules based on these classes of inconsistencies. This paper presents part of our classification scheme of design inconsistencies and describes an inconsistency identification mechanism specific to UML design models.

Section 2 describes the classification scheme and examples of design inconsistencies. Section 3 introduces production systems, and defines selected UML constructs in production system and design inconsistency rules. Section 4 describes the architecture of the implementation, evaluates and compares the rule-based approach to other research results. Section 5 draws conclusions and summarizes future work.

## 2 Classes of Design Inconsistency

We can analyze software design consistency from two perspectives. One is the *design description*, which is concerned with maintaining a consistent representation of the design. Another is the *actual design*, which is concerned with building the actual system where inconsistencies may arise from implementing design concepts.

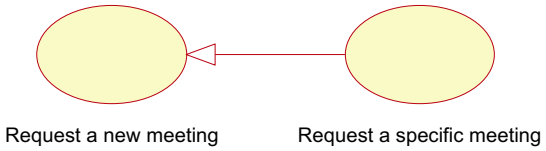
We present three classes of inconsistencies within the design description: redundancy, conformance to constraints and standards, and change. These classes cover syntactic and semantic inconsistencies of the description language, and language-independent constraints and standards. Examples are drawn for each class in UML. A detailed discussion of the classification scheme and inconsistencies within the actual design can be found in [15].

**Redundancy** One of the most frequently occurring inconsistency sources is redundancy of information. More specifically, a *redundancy* occurs when a design artifact (perhaps partial) is represented multiple times, possibly in varying views. Redundancies can occur either in design or data representation.

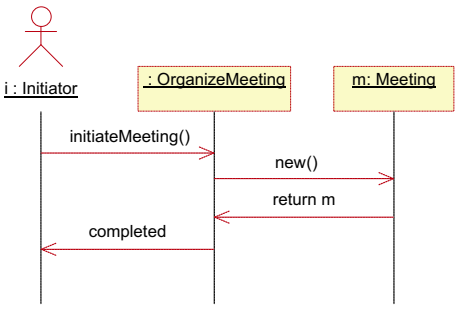
*Design related redundancy* arises when two design units have common elements, or overlap [28], in the representation. For example, a *structural redundancy* refers to an overlap between several structural and/or behavioral diagrams. It is known as *structure clash* in the context of requirements specifications [29]. Such redundancy may be desirable since it can provide additional information to a requirement specification from different perspectives, and describe the behavior of a design unit under various scenarios. For instance, in modeling feature interactions [30], two kinds of feature dependence relations can be introduced, *specialization* and *interference*. In both cases, there is redundant information.

For example, in UML, features can be expressed as use cases, which can be further elaborated using behavioral modeling constructs such as sequence diagram and collabo-

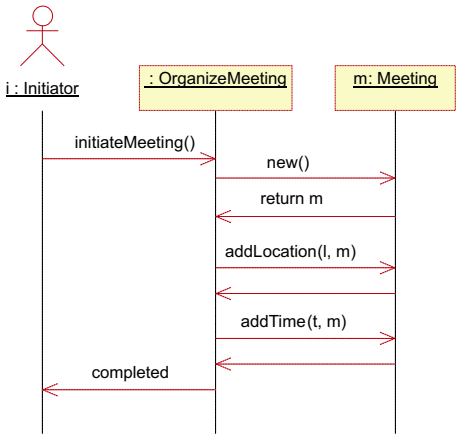
ration diagram. We define a feature, $A$, to be a specialization of another feature, $B$, if $A$ meets all of the requirements of $B$. This can be expressed using use case generalization, in which *"the child use case inherits the behavior and meaning of the parent use case; the child may add to or override the behavior of its parent; and the child may be substituted any place the parent appears"* [2]. Based on this definition, we say the design is consistent if $B$ is a subgraph of $A$. This is illustrated using the Meeting Scheduler exemplar in figures 1(a) to 1(c), where the use cases are elaborated by the respective sequence diagrams. Note that figure 1(b) is a subgraph of figure 1(c). An inconsistency occurs if the subgraph property is violated.




(a) Use Case Diagram: Request a New Meeting




(b) Sequence Diagram: Request a New Meeting




(c) Sequence Diagram: Request a Specific Meeting

**Fig. 1.** Feature Dependence Relation – Specialization

The second example shows feature interference modeled in a state diagram, where two or more features have overlapping specifications, and the states with conflicting properties may be reachable when their guarding conditions are satisfied at the same time. In figure 2, an inconsistency may occur if both features are enabled, since the destination states have contradictory properties. Standard solutions include feature prioritization.

In the above examples, information is conveyed through the integrity of multiple perspectives, and consistency can be maintained by tolerating the redundancy. Thus, over-

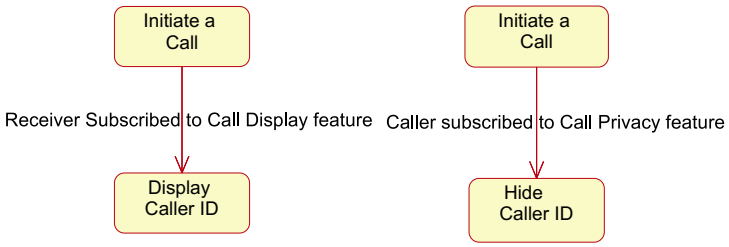



**Fig. 2.** Telephone Feature Interference

laps of design units are necessary conditions of the inconsistencies arising from redundancy, but not sufficient conditions thereof.
*Data related redundancy* usually arises from the complex relations between data. For example, a data object may be related to other objects via multiple paths in the object diagram. When changes are made to this object, all paths must be verified for validity. In some cases, removing redundant data relations can simplify the model and help keep the model consistent. Figure 3 describes the case where the relations between MeetingRoom and Coordinator are represented in both directions in an object diagram. Eliminating one of the relations makes the model simpler.

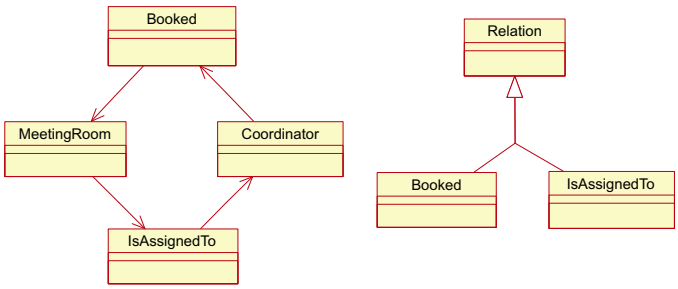



**Fig. 3.** Data Representation Redundancy in the Meeting Scheduler Example

**Conformance to Constraints and Standards** In software designs, there are many considerations outside of the system requirements. These extra-requirements include conformance to constraints [21], standards [13, 10], and patterns [12].
*Constraints* can be from intra-system conflicts, inter-system mismatches, and the modeling language. In UML, the constraints are specified by OCL [21]. Examples from the UML Foundation and Core Constraint Set are included below.

1. *The AssociationEnds must have a unique name within the Association.*
2. *No Attributes may have the same name within a Classifier.*
3. *At most one AssociationEnd may be an aggregation or composition.*

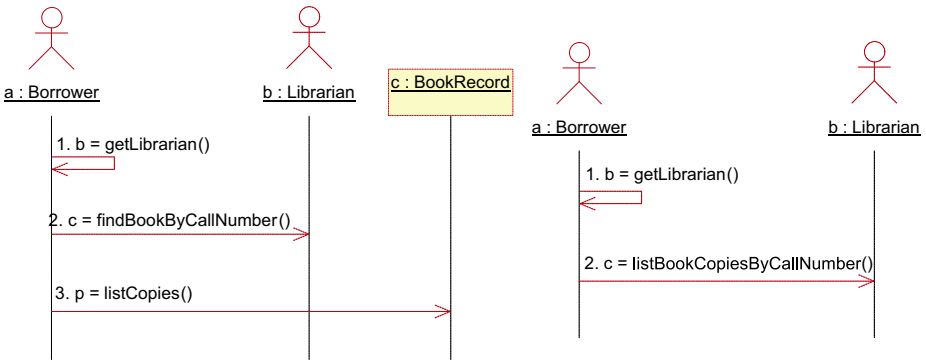
*Standards* include best practices, industry standards, and corporate standards. Conformance to standards is required in many development cultures. Nonconformance to standards raises inconsistencies, and need to be identified. For example, a well known object-oriented design standard, the *Law of Demeter*, states that "*The methods of a class should not depend in any way on the structure of any class, except the immediate (top-level)*

*structure of their own class. Further, each method should send messages to objects belonging to a very limited set of classes only*." [27]

We illustrate the violation of this design concept based on the above interpretations by a simple library example. A borrower is trying to locate copies of a book by a known author. The borrower asks the librarian to find the book by its unique call number, and then uses the record of the book to locate the copies of the book. It can be represented in Java code as the following:

```
class Borrower {···
   getLibrarian().findBookByCallNumber().listCopies();
··· }
```

The example is illustrated by a sequence diagram in figure 4(a). The violation here is that, in order to receive information provided by the `BookRecord`, the `Borrower` traverses the links to the `BookRecord` provided by the `Librarian`. The correction is shown in figure 4(b).




(a) Violation (b) Correction

**Fig. 4.** An Library Example on Conformance to the Law of Demeter

*Software design patterns* are well known both in the literature and in practice. To be able to use these, a designer must study them in depth and determine which pattern is applicable to the problem in hand. This can be a difficult task. In particular, misuses can be introduced due to misunderstanding and misinterpretation of the pattern. Therefore, we define inconsistencies that arise from misuses and violations of patterns. This requires methods to recognize the pattern and check for violation. Two examples from [12] are included below.

The *Singleton* pattern is an object creational pattern. It ensures that the designated class has only one instance and is able to provide a global access point to the instance. It can be used when multiple instances of a class are prohibited in a system, or to preclude the unnecessary object instantiations of a class. For example, a system can have many printers, but should only have one printer spooler. Moreover, the Singleton class is responsible for providing access to the reference to the instance. This pattern is violated

if a singleton class is used in the design and instances of this class are stored by other classes.

The *Façade* pattern is an object structural pattern. It is used to provide a set of interfaces to a subsystem which allows easy access to the subsystem. The intent of this pattern is to minimize communication and dependencies between subsystems in order to reduce the complexity of the whole system. Figure 5 illustrates the use of Façade pattern in a subsystem accessing model. If a class diagram of a design resembles the diagram on the left, a Façade pattern can be employed. If the use of the Façade pattern is appropriate in the design, then the absence of a façade is inconsistent with this practice.

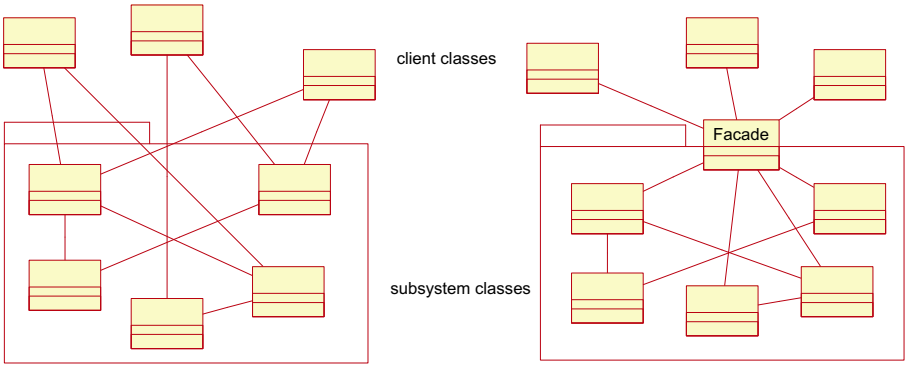



**Fig. 5.** The Façade design pattern (on the right)

**Change** A software design may undergo numerous changes before completion, due to change requests, performance tuning, and correction of environment assumptions. During the process of making design changes, inconsistencies may easily be introduced.

One source of inconsistency is in making changes using edit blocks. An edit block is a group of edit steps required to complete a desired change to the design model. If the steps are not performed as one group, it is possible that some steps may be missed, thus putting the design model in an inconsistent state.

For example, an edit block can be a design model transformation from one representation to another. When changes are made incrementally by hand, the system will remain in an inconsistent state during the entire modification process. If an unrelated change request is received during this process, then either the new request has to be put on hold until the modification is complete, or new inconsistencies will be introduced to the model. An example of such is to convert a design represented in UML to one in an architectural description language, e.g., Rapide [16].

## 3 Inconsistency Identification

In section 2, a number of classes of generally occurring design inconsistencies are presented. We observed that to identify these inconsistencies, we can define patterns or conditions that match the violations to design consistencies. Moreover, it is desirable to have automated or computer-assisted resolution to them. This leads naturally to the production system (or rule-based) solution.

In this section, we define UML constructs and inconsistency elements for production system, present rules that capture classes of inconsistencies from section 2, and define the identification and resolution process within the production system. We first review production system concepts.

A *production system* is a reasoning system that uses forward-chaining derivation techniques. It uses rules, called *production rules* or *productions* in short, to represent its general knowledge, and keeps an active memory, known as the *working memory* (WM), of facts (or assertions) which are called *working memory elements* (WMEs) [3, 11].

A *production rule* is usually written in the following form:

**IF** *conditions* **THEN** *actions*

The *conditions*, also known as *patterns*, are partial descriptions of working memory elements, which will be tested against the current state of the working memory. For example, the following rule debits a bank account.

```
IF   (transaction type:"debit" amount:x accountid:a)
     (account id:a balance:y∧{≥x})
THEN REMOVE 1
     MODIFY 2 (balance [y-x])
```

where $a$, $x$ and $y$ are variables; $\{\geq x\}$ is a test for balance$\geq x$; `REMOVE 1` deletes the first (i.e. `transaction`) WME from the working memory; and `MODIFY 2` selects the second WME and assigns the value of $y$-$x$ to `balance`.

Each condition can be either positive or negative. A negative condition is of the form -*cond*, where *cond* represents a positive condition. A rule is applicable if all of the variables can be evaluated using the WMEs in the current WM such that the conditions are met. A positive condition is satisfied if there is a matching WME in the WM; a negative condition is satisfied if there is no matching WME in the WM.

A *working memory element* has the following form,

(type attribute$_1$:value$_1$ ... attribute$_n$:value$_n$)

where type and attribute$_i$ are atoms, i.e. a string within " ", a word, or a numeral; and value$_i$ is an atom or a list within ().

The basic operation of a production system is a cyclic application of three steps until no more rules can be applied:

1. *recognize*: identify applicable rules whose conditions are satisfied by the WM;
2. *resolve conflict*: among all applicable rules (or *conflict set*), choose one to execute;
3. *act*: apply the action given in the consequent of the executed rule.

The *priority* scheme for conflict resolution is used in our method. In this scheme, a rule is chosen to execute by its preassigned priority. In case of the same priority, the first rule in order of presentation is chosen. Other schemes can be found in [3, 15].

### 3.1 Inconsistency Identification Using Production System

To identify inconsistency in UML design models, we define production rules whose conditions describe inconsistency instances, and whose consequent actions describe appropriate resolutions to these instances. By converting an UML model to and from the production system representation, we can use the production system to check for inconsistencies and resolve them appropriately. Most actions require user's input; therefore,

we use an approach similar to Argo/UML in that it delivers inconsistency notices to the user's workspace, and have the user initiate the resolution and provide input data. The details of the implementation are described in section 4.

In this section, we first define working memory elements for UML constructs and necessary information to represent inconsistencies and their resolution scheme. Next, we describe four types of rules that are defined to ensure the validity of inconsistency status:

- *inconsistency rules* that identify inconsistencies of designs;
- *resolution rules* that respond to user's choice of fixing;
- *cleanup rules* that remove expired inconsistency working memory elements and the corresponding workspace items;
- *orphan control rules* that remove the working memory elements whose parent WME either is invalid or has been deleted.

Last, we describe dynamic controls which let the user enable and disable inconsistency rules.

**Definitions for Working Memory Elements** While using the general WME representation, the composition relation between two elements (e.g., a class and its method are regarded as the parent and child respectively) is represented through the attribute `pid` in the child clause. Since each child has only one parent, but each parent may have more than one child, in this configuration, the change in the list of children will not require modification to the parent, thus minimizing maintenance of existing WMEs.

For example, the following two working memory elements describe a public class, `Meeting`, and its public attribute, `time`, of class `TimeStamp`.

```
(class     id:cls001 name:Meeting modifier:public)
(attribute id:attr001 pid:cls001 name:time classtype:TimeStamp
           modifier:public)
```

The detailed definitions of working memory elements can be found in appendix A and in [15].

For each inconsistency rule, if the condition is satisfied, one or more inconsistency WMEs are added to the working memory. The `inconsistency` type defines the main inconsistency working memory element, which indicates the occurrence of an inconsistency in a design and the location information of the involved design elements. The `userchoice` type provides resolution options to the user. The `userinput` type represents input from the user to a selected resolution option.

**Inconsistency Rules** In this section, we present examples of production rules that capture the inconsistencies identified in section 2. The formalization of the conditions of each rule describes the violation of a consistency requirement. The consequent action of a rule will add an inconsistency WME to denote the occurrence of the inconsistency, and resolution options including add/remove/modify working memory and customized function calls. Without loss of generality, the inconsistency rules below are defined with only a description message and locations of the inconsistency. Domain specific resolutions can be added to individual rules.

*Feature Dependence – Specialization* In describing features using Sequence Diagrams, if feature $A$ is a specialization of feature $B$ illustrated in the corresponding diagrams,

then an inconsistency occurs if a message or object that appears in $B$'s diagram, is absent from that of $A$. Note that in some instances, this inconsistency may be considered spurious. (`newId()` is a function that generates unique strings.)

**Rule 1** *An object is absent from the specialized sequence diagram.*

```
IF (sequenceDiagram    id:psd usecaseID:uid1)
   (sequenceDiagram    id:csd usecaseID:uid2)
   (usecaseAssociation id:a type:generalization parent:uid1
                       child:uid2)
   (sequenceObject     id:so pid:psd name:n)
   -(sequenceObject    pid:csd name:n)
THEN ADD (inconsistency id:[newId()] ruleid:"fd-1"
         location:((sequenceObject so) (sequenceDiagram psd)
         (sequenceDiagram csd))
         msg:"An object is missing from the sequence diagram of
         the specialized use case.")
```

*Feature Interference* When two features have overlapping specifications, conflicting states may be reached simultaneously.

**Rule 2** *Conflicting states reachable in state diagrams.*

```
IF (state      id:sta name:n1)
   (state      id:stb1∧{≠sta} name:n2 specification:B)
   (state      id:stb2∧{≠sta} name:n3 specification:{⇒ ¬B})
   (transition id:a from:sta to:stb1)
   (transition id:b from:sta to:stb2)
THEN ADD (inconsistency id:[newId()] ruleid:"fi-1"
         location:((state sta) (state stb1) (state stb2))
         msg:"Conflicting states occur simultaneously in State
         Diagrams.")
```

*UML Constraints* Defined in OCL.

**Rule 3** *No Attributes may have the same name within a Classifier.*

```
IF (attribute id:a1 name:z pid:t)
   (attribute id:a2∧{≠a1} name:z pid:t)
THEN ADD (inconsistency id:[newId()] ruleid:"uml-1"
         location:((attribute a1) (attribute a2))
         msg:"Attributes must be unique within a class.")
```

*Standards Conformance Rules* These rules are defined to ensure specific design standards are followed in the design model.

**Rule 4** *A design model should obey the Law of Demeter.*

```
IF (sequenceMessage id:m1 from:T1 to:T2 return:c pid:p)
   (sequenceObject  name:c type:T3∧{≠T2} pid:p)
   (sequenceMessage id:m2 from:T1 to:T3 pid:p)
THEN ADD (inconsistency id:[newId()] ruleid:"sc-1"
         location:((sequenceMessage m1) (sequenceMessage m2)
         (sequenceObject c))
         msg:"Violation of the Law of Demeter.")
```

*Pattern Recognition Rules* The condition of a pattern recognition rule formalizes one distinctive characteristic of the pattern and describes the violation of its usage.

**Rule 5** *When a Singleton pattern is used in a design, no other class objects should keep a reference to the singleton class object.*
A Singleton pattern is recognized if the class has a *static* method returning an instance of the class and a *static* attribute that stores instances of this class.

```
IF (class     id:c1 name:cn1)
   (method    id:m pid:c1 return:cn1 modifier:"static")
   (attribute id:a1 pid:c1 classtype:cn1 modifier:"static")
   (attribute id:a2 pid:c2∧{≠c1} classtype:cn1)
THEN ADD (inconsistency id:[newId()] ruleid:"pr-1"
         location:((class c1) (method m) (attribute a2))
         msg:"Reference to the object of a singleton class
         should not be stored in any other classes.")
```

**Resolution Rules** Resolution rules are used to automatically resolve inconsistency identified in design models upon receiving user choices after an inconsistency notice is delivered. In general, a resolution rule corresponds to a single user choice in response to an inconsistency notification, therefore, each inconsistency rule may have multiple resolution rules defined. We demonstrate the definition and application of resolution rules on two inconsistency rules which have well defined actions. Rules 6 and 7 are resolutions to UML Constraint - Rule 3.

**Rule 6** *Modify the name of the Attribute.*

```
IF (attribute     id:x name:z)
   (inconsistency id:s)
   (userchoice    pid:s action:modify targetID:x
                  targetType:attribute attribute:name)
   (userinput     pid:s action:modify targetID:x
                  attribute:name value:v)
THEN MODIFY 1 (name v)
     REMOVE 2,3,4
```

**Rule 7** *Remove the Attribute.*

```
IF   (attribute     id:x name:z)
     (inconsistency id:s)
     (userchoice    pid:s action:remove targetID:x
                    targetType:attribute attribute:name)
     (userinput     pid:s action:remove targetID:x)
THEN REMOVE         1,2,3,4
```

**Cleanup Rules** Cleanup rules are used to remove expired inconsistency working memory elements from the WM, thus keeping the inconsistency messages valid with respect to the current model. For every inconsistency rule, there is an associated cleanup rule. This rule checks whether the negation of the antecedent of the corresponding inconsistency rule is true, and whether the inconsistency WME exists in the WM. If both are true, the cleanup rule may be executed to remove the infeasible inconsistency working memory element. Here we show just one such rule. Rule 8 is the cleanup rule for Rule 3. Cleanup rules for the other inconsistency rules can be written in the similar fashion.

**Rule 8** *Remove inconsistency WME if no Attributes have the same name within a Classifier. (Note: ∋ tests for set membership.)*

```
IF   (attribute     id:x name:z pid:t)
     (attribute     id:y name:w∧{≠z} pid:t)
     (inconsistency location:{∋(attribute x)}∧{∋(attribute y)}
                    ruleid:"uml-1")
THEN REMOVE 3
```

**Orphan Control Rules** The production algorithm does not allow recursive removal of WMEs. Therefore, upon the execution of a rule, orphan WMEs can be left in the WM. An *orphan* is a piece of working memory element that has a parent pointer (via *pid*) linking to a non-existing WME in the WM. As we rely on the hierarchical structure of the WMEs to express the relationship of various UML design elements, the validity of the inconsistency WMEs must be maintained. To achieve this, we define orphan control production rules to remove orphan WMEs.

Rule 9 below describes orphan removal rules for `userchoice` inconsistency WMEs. Similar orphan control rules can be defined for all working memory elements.

**Rule 9** *Remove all user choice orphans.*

```
IF   -(inconsistency id:s)
     (userchoice     pid:s)
THEN REMOVE 2
```

**Dynamic Controls** Above we have described the production rules used to update the working memory elements to reflect the changes in design knowledge base and inconsistency status. Another use of production rules is to modify the applicability of existing rules on the fly. This allows the user to enable or disable production rules during the design process. To achieve this, each of the inconsistency rules described above must be modified to include a control specification in the conditions. This specification has the following form:

`-(rule id:<`*unique constant*`> disabled:yes)`

Each rule is identified by a unique constant. If the associated control specification is present in the working memory, the antecedent of the rule is unsatisfied which causes the rule to be inapplicable in the current WM.

For example, the conditions of rule 2, become the following:

```
IF -(rule      id:"fi-1" disabled:yes)
   (state      id:st_a name:n_1)
   (state      id:st_b1∧{≠st_a} name:n_2 specification:B)
   (state      id:st_b2∧{≠st_a} name:n_3 specification:{⇒ ¬B})
   (transition id:a from:st_a to:st_b1)
   (transition id:b from:st_a to:st_b2)
```

## 4 Implementation

In this chapter, we describe the current implementation, RIDE (Rule-based Inconsistency Detection Engine), by presenting the architecture of the system, and analysis of the method.

### 4.1 Architecture

RIDE is a Java implementation which can be integrated into an existing UML Design Environment, such as Argo/UML Editor [1] and Rational Rose [26]. It uses Jess [14] — an off-the-shelf Java Rule Engine that implements the RETE algorithm, to execute production rules. We plan to integrate it with Argo/UML Editor to perform on-the-fly inconsistency checking of UML models.

The architecture of the RIDE system is illustrated in Figure 6.

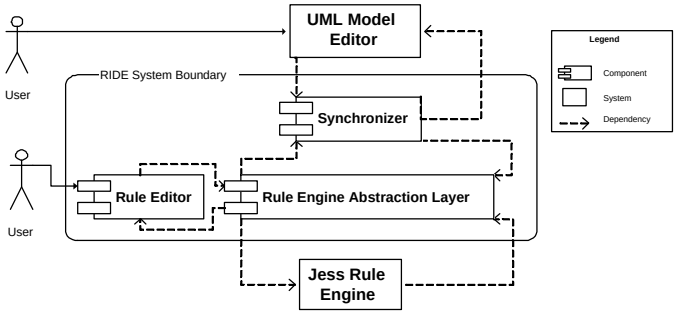



**Fig. 6.** The Architecture of RIDE

There are three components in RIDE:

**Synchronizer** As both the editor and the rule engine maintain their own representations of the UML model and inconsistency report, a Synchronizer exists to keep them identical. It sends changes of the editor's model to the rule engine via the Rule Engine Abstraction Layer, and delivers inconsistency report and modifications due to resolution by the rule engine back to the editor. Synchronizer can be thought of as an editor specific plug-in because it understands the editor's data representation and change notification mechanism.

**Rule Engine Abstraction Layer** This component allows replacing Jess by another rule engine.

**Rule Editor** This component provides a user interface which allows the user to manage the rule base. The user can display status of rules, and add/delete/modify rules.

### 4.2 Discussion

In this section, we discuss the time complexity of RIDE, and compare it with Argo, *xlinkit*, and design guidance approach. Our analytical evaluation of RIDE is based on RETE Algorithm used by the rule engine, Jess, since it performs the core computation.

The best case time complexity of one *recognize*, *resolve*, and *act* cycle is $\mathcal{O}(1)$, and the worst case complexity is bounded by either $\mathcal{O}(W^{2C-1})$ or $\mathcal{O}(P)$ [11], where $C$ is the number of patterns in a rule, $P$ is the number of rules in RETE network, and $W$ is the number of elements in working memory. This is comparable to the worst case time complexity of *xlinkit*, in which an evaluation function for each rule has worst case time complexity of $\mathcal{O}(n^k)$, where $n$ is the size of the node set of a rule, and $k$ is the maximum level of quantifier nesting [17, 18].

*Xlinkit* uses an XML-based [4] first-order logic rule language in defining consistency rules and verifying consistency of XML documents, in particular, UML design models represented in XMI [22] format. The rule language uses XPath [7] to select a set of elements as the domain of each rule and apply the subsequent conditions to these elements. Each rule expresses a desirable consistency property. If violated, an inconsistency link is added to the link base, reflecting the inconsistent elements; if satisfied, a consistency link can be added instead. Both links are represented in the XLink language [9]. When applying to UML models, the rules are defined for the UML Foundation/Core constraints described in OCL.

The *xlinkit* approach is declarative and iterative. A rule in *xlinkit* usually starts with a $\forall$ clause to characterize a particular type of element, and must be iterated over every element in its domain. On the other hand, RIDE focuses on identifications of patterns of violation to consistency rules. Since most UML documents are largely consistent, in practice, each inserted WME matches only limited number of patterns, hence minimizes the occurrence of the worst case time complexity. The rule engine only selects one rule to fire in response to each change in the working memory, instead of iterating through the entire set of production rules. Each execution is acting on a relatively small set of data; thus, the result is returned incrementally allowing more accessibility. Moreover, RIDE is integrated into an UML editor; thus the pattern matching and rule execution will be running at the background in real-time, parallel to the user's edits. Adding feedback to the workspace silently and incrementally also minimizes interruptions for the user. On the other hand, *xlinkit* is a static checker. Each execution must go through the entire rule base before any results are returned. Since the design process is to make changes to the model, RIDE has the advantage of providing just-in-time, non-interruptive feedback to the user.

The design guidance approach of Cass et al. uses a design environment that integrates a process model with *xlinkit* to perform incremental consistency checks on UML design models in parallel with the design process [6]. The process model, defined as a hierarchy of design steps using the process modeling language Little-JIL [5], provides guidance in selecting which rules to apply at each design step, and uses *xlinkit* to check the consistency of the UML model in hand against the selected rules. As a result, fewer irrelevant inconsistency messages will be returned to the user. However, it limits the set of rules that can be applied at each step, based on predetermined heuristics, which might not be optimal in general. In our approach, the rules may be arranged by priority, but no limitation on applicability is placed a priori.

In their approach, Cass et al. focused on traceability consistency between requirements and design elements. However, this approach requires a well-defined process model to be in place before the user can take advantage of automated consistency checking. This requires maintenance effort on the process model and it is not clear how this approach would deal with deviations from the predefined process model (which can also be considered as a type of inconsistency [8]). Their approach has the same worst-case time complexity as *xlinkit*, and like *xlinkit*, it does not provide automated resolutions to inconsistencies.

Another nominal contribution in this area is the work on Argo/UML Editor — an application of intelligent user interface. The editor provides an integrated UML model-

ing tool which allows the user to create and edit UML models, and provides feedback on the model through critics [24]. Critics can perform analysis on issues of correctness, completeness, consistency, etc. The feedback is delivered to the user as a To-Do item in a designated To-Do List in the workspace. The editor has two background threads that work in parallel during the modeling process. The *critique thread* selects a critic from critic waiting queues, whose definition is most closely related to the current update in design. The *invalid feedback removal thread* periodically makes a pass through the To-Do List, and verifies if the item is still valid by reapplying the critic on the current model. If the item returns as a valid critique, it will remain in the To-Do List; otherwise, it is removed from the list. Corrective automation of critiques is implemented through Wizards [25].

The main focus of research on Argo/UML is the development of an intelligent user interface. Although it offers nine different types of design critiques, it does not provide strong coverage of each type of critique. In particular, inconsistencies are treated as a generic problem of finding contradictions within the design [24]. The critiques are merely procedural descriptions of design heuristics. However, our approach offers a formal description language for inconsistencies ranging from the syntax and semantics of the UML language, to general language-independent constraints and standards. The resolutions to inconsistencies offered by RIDE can be implemented as simply as manipulation of working memory elements, or as complex object-oriented programs.

An important weakness in Argo is in selection of which critiques to apply when. The user can enable and disable groups of critics manually, but this puts the responsibility of ensuring the relevance of critiques on the end user. In addition, the current implementation has a loosely defined relation between critics, edit action types, and user/goal models such that eventually all enabled critics will be cycled [24]. This may result in an excessive number of irrelevant critiques being applied after each edit action. In our approach, the applicability of rules is determined automatically by the rule engine, which requires no action from the user, but options are provided to disable undesirable rules.

## 5 Conclusions and Future Work

In this paper, we introduced a typology of inconsistency in software design, and used this typology to define production rules for a production-system-based inconsistency management tool. This tool can be integrated into a UML design environment to provide on-the-fly inconsistency checking and resolution while UML models are edited.

In our framework, the application of rules is automatically controlled via *conflict set* resolution, and the dynamic controlling mechanism allows rule manipulations on the fly. This improves upon other approaches such as Argo/UML, which offers only manual controls to the user for enabling groups of critics [24], and *xlinkit*, where no control is available to the user [17]. As it runs, the production system can create and modify persistent working memory items, including the UML model itself. This provides a number of advantages:

– Rules can automatically modify the UML model to resolve certain inconsistencies;

- Rules can modify the behavior of other rules, by changing the contents of working memory. Such modification may range from simply enabling/disabling other rules, to meta-rules that look for patterns of occurrences of chronic inconsistency;
- Rules can detect and monitor inconsistencies that are not resolved immediately, by putting the appropriate entries into working memory;
- Sets of rules can effect complex multi-step resolutions, such as converting a data model to normal form.

The worst case time complexity of this method is comparable to that of *xlinkit*. As our detection steps are carried out incrementally in parallel with editing tasks, we expect that time complexity is less relevant than for batch checking approaches. Our current prototype implementation, RIDE system, uses an off-the-shelf rule-engine, Jess. We are continuing working on the integration with Argo/UML tool.

There are two general directions for future work. One is to continue the study of specific inconsistency classes of both the design descriptions and actual designs, and their standard solutions. Another is to observe the pattern of inconsistency occurrences and use production system to analyze the cycle of pattern and provide automated resolution.

## References


1. ArgoUML. v0.8.1a. `http://www.argouml.com`.
2. Grady Booch, James Rumbaugh, and Ivar Jacobson. *"The Unified Modeling Language User Guide"*. Addison Wesley, 2000.
3. Ronald J. Brachman and Hector J. Levesque. "Knowledge Representation and Reasoning". In preparation, 2001.
4. T. Bray, J. Paoli, C. M. Sperberg-McQueen, and E. Maler. "Extensible Markup Language (XML) Version 1.0". `http://www.w3.org/TR/2000/REC-xml-20001006`, October 2000. Second Edition. Recommendation, World Wide Web Consortium.
5. Aaron G. Cass, Barbara Staudt Lerner, Stanley M. Sutton, Eric K. McCall, Alexander E. Wise, and Leon J. Osterweil. "Little-JIL/Juliette: A process definition language and interpreter". In *The International Conference on Software Engineering*, Limerick, Ireland, 2000.
6. Aaron G. Cass and Leon J. Osterweil. "Requirements-based design guidance: A process-centered consistency management approach". Technical report, University of Massachusetts, Department of Computer Science, March 2002.
7. J. Clark and S. DeRose. "XML Path Language (Xpath) Version 1.0. Recommendation". `http://www.w3.org/TR/1999/REC-xpath-19991116` World Wide Web Consortium, November 1999.
8. Gianpaolo Cugola. "Tolerating Deviations in Process Support Systems via Flexible Enactment of Process Models". *Transactions on Software Engineering*, 24(11):982–1001, 1998.
9. S. DeRose, E. Maler, D. Orchard, and B. Trafford. "XML Linking Language (Xlink) Version 1.0.". `http://www.w3.org/TR/2000/CR-xlink-20000703` World Wide Web Consortium, Candidate Recommendation, July 2000.
10. Wolfgang Emmerich, Anthony Finkelstein, Carlo Montangero, Stefano Antonelli, Stephen Armitage, and Richard Stevens. "Managing Standards Compliance". *IEEE Transactions on Software Engineering*, 25(6):836–851, 1999.
11. Charles L. Forgy. "Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem". *Artificial Intelligence*, 19(1):17–37, 1982.

12. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *"Design Pattern Elements of Reusable ObjectOriented Software"(Foreword by Grady Booch)*. Addison-Wesley, 1995.
13. ISO. "What are standards?". `http://www.iso.ch/iso/en/aboutiso/introduction/index.html`, 2002.
14. Jess. v6.0. `http://herzberg.ca.sandia.gov/jess/`.
15. WenQian Liu. "Rule-based Detection of Inconsistency in Software Design". Master's thesis, University of Toronto, Department of Computer Science, July 2002.
16. David C. Luckham. "Rapide: A Language and Toolset for Simulation of Distributed Systems by Partial Orderings of Events". Paper presented at DIMACS Partial Order Methods Workshop IV, Princeton University, 1996.
17. Christian Nentwich, Licia Capra, Wolfgang Emmerich, and Anthony Finkelstein. "xlinkit: A Consistency Checking and Smart Link Generation Service". ACM Transactions on Internet Technology. To appear. Available at `www.cs.ucl.ac.uk/staff/A.Finkelstein/papers`, 2002.
18. Christian Nentwich, Wolfgang Emmerich, and Anthony Finkelstein. "Checking Distributed Software Engineering Content". Technical Report RN/01/11, UCL Department of Computer Science, 2001.
19. Bashar Nuseibeh, Steve Easterbrook, and Alessandra Russo. "Leveraging Inconsistency in Software Development". *Computer*, 33(4):24–29, 2000.
20. Bashar Nuseibeh, Steve Easterbrook, and Alessandra Russo. "Making inconsistency respectable in software development". *The Journal of Systems and Software*, 58(2):171–180, 2001.
21. OMG. "Unified Modeling Language (UML) Specification", March 2000.
22. OMG. "XML Metadata Interchange (XMI) Specification 1.1", November 2000.
23. Jason E. Robbins, D. M. Hilbert, and D. F. Redmiles. "Argo: A design environment for evolving software architectures". In *The Nineteenth International Conference on Software Engineering*, pages 600–601, 1997.
24. Jason E. Robbins and David F. Redmiles. "Software Architecture Critics in the Argo Design Environment". *Knowledge-Based Systems*, 11(1):47–60, 1998.
25. Jason Elliot Robbins. "Design Critiquing Systems". Technical Report UCI-98-41, University of California, Irvine, 1999.
26. Rational Rose. `http://www.rational.com/rose`.
27. Markku Sakkinen. "Comments on "the Law of Demeter" and C++". *ACM SIGPLAN Notices*, 23(12):38–44, 1988.
28. George Spanoudakis, Anthony Finkelsteain, and David Till. "Overlaps in Requirements Engineering". *Automated Software Engineering*, 6(2):171–198, 1999.
29. Axel van Lamsweerde, Robert Darimont, and Emmanuel Letier. "Managing Conflicts in Goal-Driven Requirements Engineering". *IEEE Transactions on Software Engineering*, 24(11):908–926, 1998.
30. Pamela Zave. "An experiment in feature engineering". In Essays by the Members of the IFIP Working Group on Programming Methodology, Springer-Verlag. To appear.

# A Definitions for Working Memory Elements

| *Class Diagram, Class* | | |
|---|---|---|
| Type | Attribute | Value |
| `classDiagram` | `id` | A unique identifier string. |
| | `name` | The name of the diagram. |
| `class` | `id` | A unique identifier string. |
| | `pid` | The class diagram identification to which the class is defined in. |
| | `name` | The name of the class. |
| | `modifier` | Example modifiers: `public, private, default, protected` |
| `method` | `id` | A unique identifier string. |
| | `pid` | The class id to which the method belongs to. |
| | `name` | The name of the method. |
| | `modifier` | Example modifiers: `public, private, default, protected` |
| `parameter` | `id` | A unique identifier string. |
| | `pid` | The method identifier to which the parameter is part of. |
| | `name` | The name of the parameter. |
| | `type` | The class name that the parameter is typed. |
| | `modifier` | Example modifiers: `public, private, default, protected` |
| `attribute` | `id` | A unique identifier string. |
| | `pid` | The identifier of the type to which the attribute belongs. |
| | `name` | The name of the attribute. |
| | `type` | The class name that the attribute is typed. |
| | `modifier` | Example modifiers: public, private, default, protected |

| *Association* | | |
|---|---|---|
| Type | Attribute | Value |
| `association` | `id` | A unique identifier string. |
| | `name` | The name of the association. |
| `associationEnd` | `id` | A unique identifier string. |
| | `name` | The name of the association. |
| | `pid` | The identifier of the association to which this association end belongs. |
| | `class` | The class name of the end. |
| | `type` | `generalization, realization, dependency, association` |

| *State Diagram* | | |
|---|---|---|
| Type | Attribute | Value |
| `state` | `id` | A unique identifier string. |
| | `name` | The name of the state. |
| | `specification` | The specification expression of the state as string. |
| `transition` | `id` | A unique identifier string. |
| | `from` | The state id of the origin of the transition. |
| | `to` | The state id of the destination of the transition. |

| *Sequence Diagram* | | |
|---|---|---|
| Type | Attribute | Value |
| `sequenceDiagram` | `id` | A unique identifier string. |
| | `name` | The name of the sequence diagram. |
| | `notes` | Additional notes. |
| | `associateTo` | The use case id to which the sequence diagram is associated. |
| `sequenceObject` | `id` | A unique identifier string. |
| | `pid` | The sequence diagram id to which the object belongs. |
| | `name` | The object name or variable name. |
| | `type` | The type of the object, i.e. the class name. |
| `sequenceMessage` | `id` | A unique identifier string. |
| | `pid` | The sequence diagram id to which the object belongs. |
| | `name` | The name of the message. |
| | `from` | The object id of the origin of the message. |
| | `to` | The object id of the designation of the message. |
| | `code` | `sync, async, destroy, return, new` |

| *Inconsistency Resolution Elements* | | |
|---|---|---|
| Type | Attribute | Value |
| `inconsistency` | `id` | A unique identifier string. |
| | `ruleid` | A unique name of the offending rule. |
| | `name` | The unique name of the offending inconsistency rule. |
| | `location` | List of WME type and id pairs of offending elements. |
| | `msg` | The text that describes the detail of the inconsistency. |
| `userchoice` | `id` | A unique identifier string. |
| | `pid` | The inconsistency WME identifier. |
| | `action` | The specification of the function call. |
| | `actionText` | The explaination of the action. |
| | `targetID` | The references to the identifier of the offending element. |
| `userinput` | `id` | A unique identifier string. |
| | `pid` | The inconsistency WME identifier. |
| | `action` | `add, remove, modify` |
| | `targetID` | The identifier of the offending element. |