International Journal of Computer Applications (0975 – 8887)
Volume 78 – No.14, September 2013

Consistency Management of Heterogeneous Software Artifacts

Mounir Zekkaoui
Laboratory LIST
FST-Tangier
Tangier-Morocco

Abdelhadi Fennan
Laboratory LIST
FST-Tangier
Tangier-Morocco

ABSTRACT

The development of complex software systems involves the participation of many stakeholders (e.g. customers, users, analysts, designers, and developers) who collaborate in publishing numerous heterogeneous software artifacts such as source code, analysis models and design, unit tests, XML deployment descriptors, the user guides, among many others. Since these artifacts are spread over several designs sites and therefore stored in several version managers. In this context it becomes very difficult to ensure their consistency and manage the impact of their evolution out the development process.

In this article we propose a unified meta-model to represent the different elementary operations built on a set of heterogeneous artifacts and establish a uniform formalism to express consistency rules as logical constraints based on the meta-model construction. Our approach allowing us to deal with consistency between different artifacts whatever their kind. We will validate our approach by building a eclipse plug-in (in progress) that will provide an interface to declare the rules of consistency and check engine to detect violations of constraints listed previously.

General Terms

Design, Verification.

Keywords

Artifact, Meta-model Construction, Consistency, Software Engineering, Inconsistency Rules, Construction Operations.

1. INTRODUCTION

We ask Software Engineering has been described as a discipline of description [2]. Software engineers make use of a large number of different artifacts, including source code, analysis models and design, unit tests, XML deployment descriptors, the user guides, among many others. Since these artifacts may evolve over the time through participation and collaboration of many engineers throughout the development process, [3] establishing and maintaining consistency among descriptions presents several problems:

- descriptions vary greatly in their formality and precision;
- individual descriptions may themselves be ill-formed or self-contradictory;
- descriptions evolve throughout the life cycle at different rates; and
- checking consistency of a large, arbitrary set of descriptions is computationally expensive.

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 [16]. 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 [3].

Spanoudakis and Zisman [1] define six inconsistency management activities that should be undertaken. The first activity, inconsistency detection, is of special interest as it defines the foundation of the whole process. Considering this activity, two families of approaches are identified: the logic-based approaches and the model checking approaches. The logic-based approaches are defined by the use of some formal inference techniques to detect any kind of model inconsistency. The model checking approaches deploy dedicated model verification algorithms that are well suited to detect specific behavioural inconsistencies but are not well adapted to other kinds of inconsistencies.

The approach called « consistency management » believes that it is impossible to ensure the global consistency of all software artifacts at all times. Any artifact can be temporarily inconsistent. The main problem of this approach lies in tracking inconsistencies. It is necessary to detect the introduction of new inconsistencies and removing existing inconsistencies in successive changes made by developers on artifacts, without impeding the progress of the development process.

It is important to note that current approach do not in general work on homogeneous artifacts (eg model objects [6, 7, 9, 11, 13, 14 et 15]), or using pivots formats (such as XML [4 et 10]) to hide the heterogeneity. Moreover, they cannot generally cope with the evolution of different heterogeneous artifacts.

In this article, we present CMAC, our approach to managing consistency which has the particularity of being based on construction operations of software artifacts. CMAC detects the presence or absence of inconsistency artifacts. Inconsistencies are specified by logical rules on construction operations. This representation has the advantage of supporting the implementation of incremental detection providing performance gains very interesting. Moreover, it allows the definition of methodological rules of inconsistency to specify temporal orders between construction operations.

The remainder of this paper is structured as follows. Section 2 describes, accurately, how we attacked the problem, the methods and tools we used and how we did (the meta-model construction, unified formalism for managing inconsistency).

Section 3 presents the results of our approach (the prototype we built) and we conclude in the last section.

2. METHODOLOGY

Understanding of the software and the acquisition of knowledge about the system are essential for all activities in software engineering. The term artifact means any entity falling within the scope of software development. However, it is both difficult and complex to identify the data inconsistency across all artifacts starting from the artifact changed. A change to any software artifact must be taken into account and it will treat by controlling the consistency rules defined in relation to other artifacts. It is then necessary to dispose an abstract and unified representation of software artifacts to facilitate the expression and management of consistency between these heterogeneous artifacts.

To check the consistency rules, we proposed CMAC approach that identifies the elements that do not comply with the consistency rules of artifacts. CMAC is composed a meta-model of structural and unified representation of the different software artifacts and a uniform mechanism for expressing consistency rules in these artifacts.

2.1 Artifact construction

We considered that all artifacts, regardless of their type, are comparable to typed graphs [17]. An Artifact is then composed of elements. Each element is typed. It can carry values for attributes and can reference other elements. All attributes and references are also typed. The artifacts are interrelated them at different levels of granularity [18]. All artifacts can be represented according to the hierarchisation in levels by the pair $< \sum_{Lv} , \sum_{art} >$ where $\sum_{Lv}$ is the set of all levels. The k^{th} artifact of the j^{th} level is represented by the pair $< Lv_j , art_k >$. For example, $<Lv_{Class}$, ***Calculator***$>$ denotes the fact that Calculator is an artifact belonging to the class level.

It is important to note that the current approaches extract the data either by browsing all the artifacts with a listening on construction operations of developers at the end of to store in a database [5, 7, 8, 11 et 12], or through the conversion of all artifacts into XML before applying the consistency rules [4].

Extract all artifacts and listening different construction operations is a very heavy process to install it in a development environment. In our approach, we allow users (who are often responsible for the development) to extract artifacts by level who need and stored in the database at the end of applying different consistency rules (eg classes, methods, attributes, beans deployment descriptor).

For more complex artifacts such as compounds artifacts, texts files and documents of business rules. We adopt an approach that is defined manually by the expert of evolution. This means that artifacts are extracted manually or by specific algorithms implemented by artifact level such as java and xml files.

The figure (see Figure 1) shows the meta-model we proposed, to present the structure of uniform representation of artifacts. Of specific algorithms defined by artifacts level and other added manually by experts following the approach adopted. The event listener adapted for each type of artifacts are listening to the construction operation (creation, modification, deletion) to represent all artifacts in meta-model, specifying temporal orders between construction operations, for better control of software evolution.

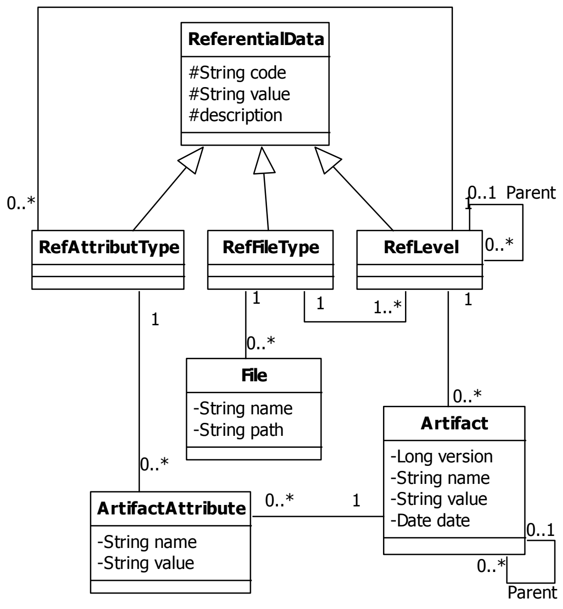


Fig 1: Meta-model of Artifact construction

We tried to provide a simple and scalable meta-model to represent all types of artifacts.

We considered that all artifacts are stored in files. Each file type must have hierarchical abstro-granular levels (eg class level which contains methods which can find the parameters). Artifacts can also be represented hierarchically, an artifact can have elements (artifacts), and each element can have other elements (artifacts), and so on. Each artifact belongs to a level and may have other attributes by level set manually by experts.

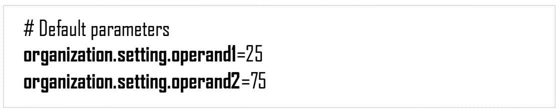
Taking the following example, a class java « *Calculator.java* » (see Figure 2) is referenced in the deployment file « *app-context.xml* » (see Figure 3), the deployment file also contains references to parameters in a property file « *build.properties* » (see Figure 4).

```
package ma.organization.calculation;
public class Calculator {
    private int operand1;
    private int operand2;
    public Calculator(int pOperand1, int pOperand2) {
        this.operand1 = pOperand1;
        this.operand2 = pOperand2;
    }
....
}
```

Fig 2: Sample java file “Calculator.java”

```
<?xml version="1.0" encoding="UTF-8"?>
<beans ...>
  <bean id="calculator" class="ma.organization.calculation.Calculator" >
    <constructor-arg value="${organization.setting.operand1}" />
    <constructor-arg value="${organization.setting.operand2}" />
  </bean>
</beans>
```

Fig 3: Sample xml context file “app-context.xml



```
# Default parameters
organization.setting.operand1=25
organization.setting.operand2=75
```

Fig 4: Sample properties file "build.properties"

Assuming that the development engineer wants to monitor the consistency between:

1. The declaration of beans in the deployment descriptor and the existence of these classes in the same path specified in the xml file.
2. The attributes of beans declared in the deployment descriptor and attributes of the class itself
3. References used in the deployment file and the existence of these parameters in the properties file.

To meet these rules consistency, we propose a unified approach to extract only the artifacts which will need it, that means the developer responsible for the evolution specifies the different artifacts to extract by level, and it is through a form choosing the file type and artifacts levels to extract. For complex files (not supported by the application) such as text files and documents, specific algorithms can beings developed following the adopted approach end to enrich the form data.

The Tables (see Table 1) and (see Table 2) represent the different levels and artifacts they may be extracted from the previous example at the end of meet consistency rules.

Table 1. Artifacts level extracted from the example

value	parent	fileType
package		java
javaclass	package	java
attribut	javaclass	java
constructor	javaclass	java
parameter	constructor	java
xmlbean		xml
xmlclass	bean	xml
constructor-arg	bean	xml
paramKey		properties
paramValue	paramKey	properties

Table 2. Artifacts extracted from the example

value	parent	refLevel
ma.organization.cal culation		package
Calculator	ma.organiz ation.calcul ation	javaclass
operand1	Calculator	attribut
Operand2	Calculator	attribut
Calculator	Calculator	constructor
pOperand1	Calculator	parameter
pOperand2	Calculator	parameter
bean		xmlbean
ma.organization.cal culation.Calculator	bean	xmlclass
${organization.setti ng.operand1}	bean	Constructor-arg
${organization.setti ng.operand1}	bean	Constructor-arg
organization.setting .operand1		paramKey
organization.setting .operand1		paramValue

2.2 Engine of consistency rules

Several classifications of consistency rules have been provided in [1] and others. Our goal is not to define a new classification of consistency rules, but rather to provide a uniform mechanism (engine of rules) for dealing with artifacts inconsistency regardless of their types.

For the consistency rules, we proposed to define them as of relationships between the elements defined in the meta-model construction (chapter 2.1). In the case of our model, we allow users (who are often responsible for the development) to define dependencies between software artifacts. These relationships are represented as of logical formulas defined by the development engineer through an intuitive interface (Eclipse plugin under construction). For more complex relationships we adopt a specific language that is to define them manually by the expert of evolution.

Modeling the inter-relationships artifacts is a complex and very important task. We consider three types of relationships (see Figure 5). They are:

1. **Inter-files relationships:** These relationships connect artifacts belonging to two different file systems. This is the case for example of the relationship between a UML class and a Java class that implements or the relationship between java class and the deployment descriptor (xml file).
2. **Horizontal relationships:** they represent different kinds of semantic links in the same file and linking artifacts of the same granular level. This is particularly the case of the call relationship between two methods or the inheritance relationship between two classes, ...
3. **Vertical relationships:** they connect two artifacts belonging to the same file at different granular levels. An example of this type of relationship is the one between a class of these attributes, a method body or block the instructions that compose it, ...

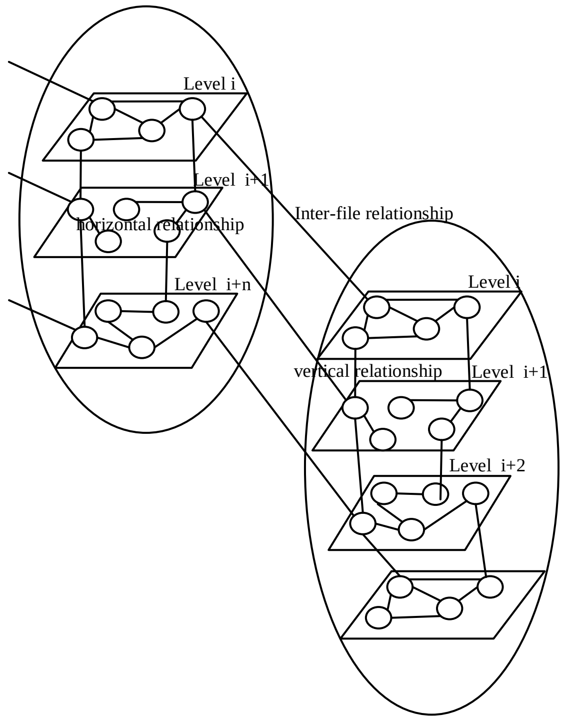


Fig 5: Classification of artifact relationships

Figure 6 presents our proposal of a meta-model of construction and consistency rules between the different artifacts for monitoring the impact of their evolution (detect violations of methodological constraints)

We considered that a consistency rule « *Rule* » is composed of several conditions « *Condition* ».

A condition can be either:

1. A composition of several sub condition « *Condition* ».
2. Or in the form of two parameters « *ParameterA & B* » and one operation « *RefOperation* ».

« *ParameterA* » may have the exact value of artifact or after the application of the method « *RefMethode* ».

« *ParameterB* » same principle as the « *ParameterA* », except that it can also have a value entered by the user.

« *RefMethod* » specific methods that can be applied sue parameters (eg StringToInt, NumberOfChart and others).

« *RefOperation* » can have the value of the following symbols (>, <, = =, equals and others), usually all the signs used in the "if" statement of the Java language.

(e.g. the rule *"(StringToInt("23")> 3)"*, the sign ">" represents the « *RefOperation* », "*23*" is the value of « *Artifact* », "*StringToInt("23")*" is the value of « *ParameterA* » and "3" is the value of « *ParameterB* »).

The user can specify an error message « *Message* » in case of inconsistency, as he may specify the severity level for each rule (in case of "*BLOCKING*" the system stops the application if it is started).

« *RuleState* » containing a state history of each rule (*successful* and *not succesful*) together with the current state (*historized: false*), this table is powered to each inconsistency check.

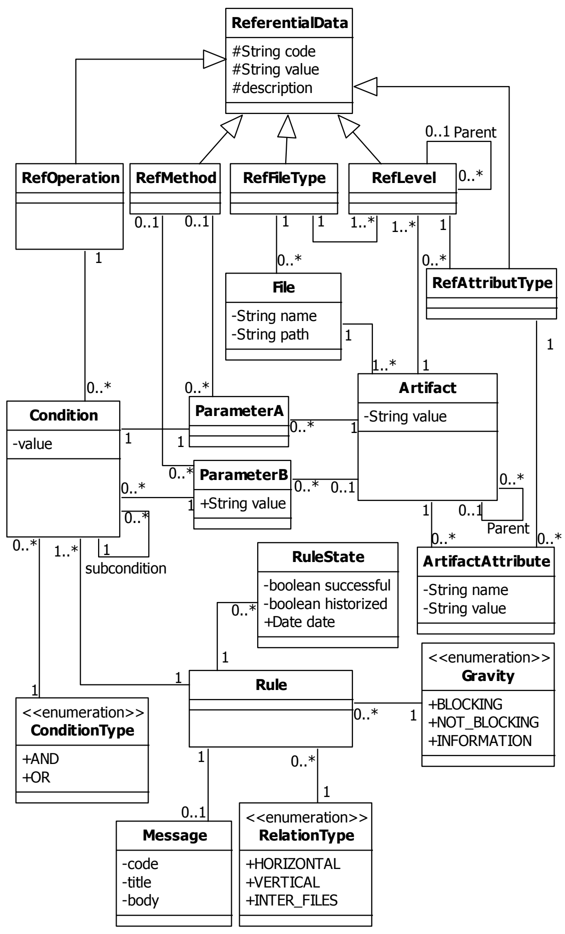


Fig 6: Meta-model of construction and consistency rules

2.3 Detection of inconsistency

We have implemented a system for the automatic verification of inconsistency. The program is developed with Java language, it is based on the extracted artifacts and consistency rules defined by the developer before displaying the different specific messages to each inconsistency in the console. The key principle is to convert the rules defined in the meta-model to queries written in Java language.

A consistency rule is usually written in the following form :

`IF` ***conditions*** `Then` ***actions(A)*** *Else* ***actions(B)***

A condition can be either a set of other conditions

condition = $\sum_{\$condition}$

Or may have the following form :

Condition **=** *$ConditionType* ***(****$ParameterA*
*$RefOperation $ParameterB****)***

- ***$ConditionType :*** can have two values {OR | AND}.
- ***$RefOperation :*** can have the following symbols *{<, >, =, ==, !=, equals, ...}* almost all the symbols used in the ***if*** condition of Java language.

- ***$ParameterA :** $RefMethode($Artefact)*, is the value of an artifact, be simple or well after application of a specific method (e.g. converting a string to a numeric value).
- ***$ParameterB :** {Value | $RefMethode($Artefact)}* the same principle as *$ParameterA*, except that *$ParameterB* can have a value entered by the developer.

***actions(A)** = (update current $RuleState) and (add new $RuleState)*

If the conditions of the rule are true then → update the current state *$RuleState(historized:false)* of the rule by changing the value of the attribute *historized* to *true*, then add a new object *$RuleState* with *successful:true*, *historized:false* and *date:new Date().*

***actions(B)** = (update current $RuleState) & (add new $RuleState)*

If the conditions of the rules are wrong then → 1- Update the current state *$RuleState(historized:false)* of the rule by changing the value of the attribute *historized* to *true*. 2- Add a new object *$RuleState* with s*uccessful:false*, *historized :false* and *date:new Date().* 3- if *$Gravity:blocking* → Stop processing and display specific message (or generic message if the developer has not specified an error message) in the console, else if *$Gravity:{not blocking|information}* display the message in the console without stop processing.

3. RESULTS

As a proof of concept of our approach, we have built a prototype in the Java programming language. The key idea is that the artifacts and consistency rules are represented in a unified meta-model and a Java program is based on the meta-model for check the inconsistency. This java prototype will be integrated into the Eclipse development environment (CMAC eclipse plugin in progress) and development tool *Checkstyle* (in progress). Users can trigger the inconsistency check or set the start settings (at project start, activate the listeners, ...). Of listeners per file and level artifacts are listening to different construction operations at the end update the relevant data in the meta-model (e.g. if a file has been updated → update all artifacts of this file).

3.1 Architecture

Our prototype is composed of two main components: the *"Artifact Builder"*, the "*Consistency rules Builder"* and the "*Check Engine*" (see Figure 7).

"Artifact Builder" is responsible for the following two main tasks:

1. The extraction of artifacts from different file types and levels selected by developers, and then store them in a unified database of construction.
2. Listening to the various construction operations (add, update and delete) made by the developers on the files and monitored artifacts levels in order to update the database of construction.

"*Consistency rules Builder"* allows users to specify different consistency rules through an intuitive graphical interface or through the specific language for complex rules. Also allows setting the timing of the launch of the engine of inconsistency check.

"*Check Engine*" it is responsible for detecting inconsistencies. It analyzes the rules stored in the database (conversion to the specific language before compilation) and produces an inconsistency detection report.

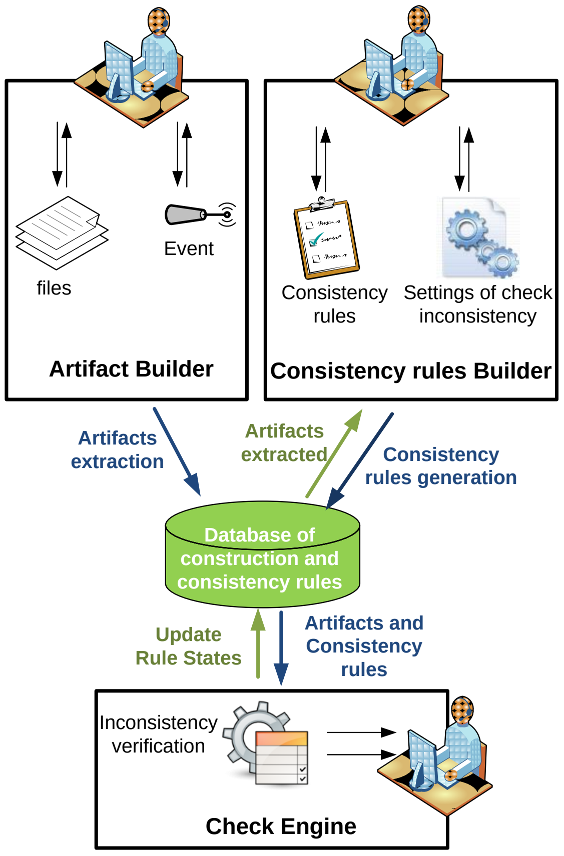

files
Event
Artifact Builder
Consistency rules
Settings of check inconsistency
Consistency rules Builder
Artifacts extraction
Artifacts extracted
Consistency rules generation
Database of construction and consistency rules
Update Rule States
Artifacts and Consistency rules
Inconsistency verification
Check Engine

Fig 7: CMAC architecture

3.2 Artifact construction Builder

We have defined two kinds of Artifact Builder. One is a *file reader* and the other is an *event listener*. The file reader can scan all file type, and outputs all the artifacts that correspond to different levels selected by the user (existing levels or implemented by the developer). The event listener can receive events raised by the various modification made by developers in order to update artifacts in the database. This enables the incremental checking of inconsistencies.

The *file reader* artifact builder has been developed in Java on top of CMAC framework using Strategy pattern. Strategy is a software design pattern, whereby an algorithm's behavior can be selected at runtime based on the type of data [19], in our case the type of data is the level of artifact. We proposed to implement a specific class by artifact level, and for more complex levels the developer can add more classes by level following the adopted approach. In each class we defined the specific algorithm for extracting and recording the artifacts in the database with corresponding levels.

The *event listener* has been also developed in Java using the *WatchService* API of Java 7. The objective is to monitor the various changes to the files and Artifacts levels, then make the call to the file reader in order to update data in the database.

3.3 Consistency rules Builder

The principle of the consistency rules builder is very simple, it is a CMAC plugin interface allowing users to define consistency rules as logical operations between artifacts values, already extracted by the *Artifact Builder*.

For the simple rules (those defined between two artifacts for example), we have proposed to define them manually through an interface proposed by the plugin, and for more complex rules, the developers must go through an option of specific language that is available in the same plugin interface.

The two ways of declaring consistency rules deliver output recorded in a unified meta-model.

Consistency rules Builder also provides a programming interface that allows users to change the default configuration particularly the launch of the *check engine*.

3.4 Check Engine

We chose to build our check engine in java language. Our approach is to convert the different registered rules to queries in java language (section 2.3), and at the output of the execution of these conditions, the program update states history of each rule and display specific messages when inconsistency.

4. CONCLUSION AND PERSPECTIVES

This paper presents two main contributions. We have presented a unified meta-model representation of different heterogeneous artifacts and a uniform formalism specification of methodological consistency rules to support information engineering in software development projects. And we validated our approach by building a system (check engine) to detect violations of methodological rules.

We have presented an implementation of this approach, called CMAC, which defines the consistency rules and follow their evolution. CMAC fulfills the requirements from the introduction. It supports artifact information retrieval and is open for user-defined custom retrievals. In doing so, it provides a single uniform means for specifying such retrievals over multiple heterogeneous artifacts, avoiding the need for the developer to get acquainted with a plethora of diverse tools. Due to its modular architecture, CMAC is customizable to project specific needs, allowing to select from the available tools and to be incrementally extended with new information processing tools.

Due to its modular architecture, CMAC serves as a good basis to implement a whole chain of processing tools on top of it. Many tasks in the context of software development and reverse engineering require not only to retrieve information of interest, but also to process it in some meaningful way. Tasks like pre-processing, refactoring, consistency validation etc., all require a custom extraction tool. For example, block modification of any entity before updating the class in the UML model. This example shows that CMAC is a good basis for the implementation of tools and languages for the development of software engineering.

We are currently working to implement the plugin CMAC implementing our approach which can facilitate the management of inconsistency between heterogeneous artifacts. Finally, we wanted to integrate our approach in several object mapping and refactoring tools, citing for example "*Checkstyle*" and "*Dozer*" tools.

5. REFERENCES

[1] Spanoudakis, G. Zisman, A. 2001. Inconsistency Management in Software Engineering: Survey and Open Research Issues, Volume 1 (2001), pp. 329-380.

[2] Jackson, M. 1995. Software Requirements & Specifications: A Lexicon of Practice, Principles, and Prejudices, Addison-Wesley, Wokingham, England.

[3] Bashar, N. Easterbrook, S. and Russo, A. 2000. Leveraging Inconsistency in Software Development.

[4] Eichberg, M. Mezini, M. Ostermann, K. and Schäfer, T. 2004. XIRC: A Kernel for Cross-Artifact Information Engineering in Software Development Environments. WCRE 2004: 182-191.

[5] Blanc, X. Mounier, I. Mougenot, A. and Mens, T. 2008: Detecting model inconsistency through operation-based model construction. ICSE 2008: 511-520.

[6] Egyed, A. 2007. Fixing Inconsistencies in UML Design Models. ICSE 2007: 292-301.

[7] Blanc, X. Mougenot, A. Mounier, I. and Mens, T. 2009. Incremental Detection of Model Inconsistencies based on Model Operations, Conference on Advanced Information Systems Engineering, Springer-Verlag Berlin, Heideberg 2009.

[8] Mougenot, A. Blanc, X. and Gervais, M.-P. 2009. D-Praxis: A Peer-to-Peer Collaborative Model Editing Framework, Distributed Applications and Interoperable Systems, 9th IFIP WG 6.1 International Conference, DAIS 2009, 2009, p. 16-29.

[9] Caffiau, S. Girard, P. Guittet, L. and Blanc, X. 2011. Vérification de cohérence entre modèles de tâches et de dialogue en conception centrée-utilisateur.

[10] Nentwich, C. Capra, L. Emmerich, W. and Finkelstein, A. 2002. xlinkit: A Consistency Checking and Smart Link Generation Service.

[11] Egyed, A. 2009. Automatically Detecting and Tracking Inconsistencies in Software Design Models.

[12] Le Noir, J. Delande, O. Exertier, D. Silva, M. A. A. and Blanc, X. 2011. Operation Based Model Representation: Experiences on Inconsistency Detection.

[13] Liu, W. 2002. RULE-BASED DETECTION OF INCONSISTENCY IN SOFTWARE DESIGN.

[14] Liu, W. Easterbrook, S. and Mylopoulos, J. 2002. RULE-BASED DETECTION OF INCONSISTENCY IN UML MODELS.

[15] Silva, M. A. A. Mougenot, A. Blanc, X. and Bendraou, R. 2008. Towards Automated Inconsistency Handling in Design Models

[16] Nuseibeh, B. Kramer, J. and Finkelstein, A.C.W. 1994. A Framework for Expressing the Relationships between Multiple Views in Requirements Specification, IEEE Trans. Software Eng. pp. 760-773.

[17] Ehrig, H. Prange, U. and Taentzer, G. 2004. Fundamental Theory for Typed Attributed Graph Transformation, Graph Transformations, Second International Conference, ICGT 2004, Springer 2004.

[18] Basson, H. 1998. An integrated model for impact analysis of software change. In International Conference On Osftware Quality Management.

[19] Erich, G. Richard, H. Ralph, J. John, V. 1993. Design Patterns: Abstraction and Reuse of Object-Oriented Design