Detecting Design Level Anti-patterns; Structure and Semantics in UML Class Diagrams

Eman K. Elsayed*, Enas E. El-Sharawy
Mathematics and Computer science Department, Faculty of science, Alazhar University, Cairo, Egypt.

* Corresponding author. Email: emankaran10@azhar.edu.eg
Manuscript submitted June 10, 2017; accepted August 8, 2017.
doi: 10.17706/jcp.13.6.638-654

Abstract: Nowadays generation of reliable patterns is a challenge in the software engineering field, so determination of the anti-patterns becomes an effective and objective concept to evaluate any design. This paper proposes a general method to detect anti-patterns; structure and semantics in case of UML(Unified modeling language) class diagram. The proposed method is classified as a hybrid between mathematical and meta-model approaches. Its four phases merge between OWL(Web Ontology Language) Ontology-based and event B for detection many anti-patterns; semantic and structure in UML class diagram components (attributes, classes, operations, and associations). The paper proves the proposed method in two ways; the first way is theoretical by coupling UML components with OWL and Event-B. The second way is experimental by applying the method on a sample of nine famous UML class diagrams used as templates. The method detects and corrects the anti-patterns which appeared 519 times.

Key words : Pattern, anti-pattern, Uml, event–B, ontology.

1. Introduction

A perfect design pattern is an important issue in software engineering. There are many advantages for using patterns in software development, such as improvement in code standards, scalable designs, and avoiding time-consumption. The ability to successfully design software projects is highly depending on using patterns without informal problems "Anti-pattern". Thousands of UML templates on the internet have Anti-patterns. This will lead us to believe that having a general tool for verification and validation of a pattern design is important. In software engineering, the levels of detection of anti-patterns are project management level, design level, and code level. Good results have been obtained in all levels. There is a tool under implementation for detecting the project management anti-patterns in the Aristotle University of Thessaloniki [1]. Also, there are many papers that propose detection techniques at the design level as proposed in reference [2], but the code level in reference [3] focuses on certain types of anti-patterns. We will present in more details a background about detection of anti-patterns at design level in related works Section 3.

No attempt has been made to create a general method based on mathematical theory to be stable and could check the structural and semantic patterns at the same time. The alternative methods that use OWL ontology- based without event-B can't detect all anti-patterns that the proposed method can do. Implicitly, discovering the anti-patterns at the design level will prevent a lot and not all anti-patterns at the code level.

This paper will explore the area of bad practices, namely anti-patterns or dark patterns, and their consequences in UML class diagram specifically at design level. The main idea of this paper is suggesting a general method to detect and correct semantic and structural anti-patterns at design level on UML class diagrams. That occurs by

integrating the advantages of using event-B and OWL ontology-based, then coupling the UML class diagram components with event-B and OWL ontology-based. This mapping is classified into four phases for offering a systematic detection. The proposed method covers all UML class diagram components; attributes, operations, classes, and association.

The proposed arrangement steps in each phase give us better solution than others. Each step can detect and may also correct some types of anti-patterns.

The rest of this paper is organized as follows; in Section 2 we will present the main concepts that will be used as static and dynamic analysis of the pattern, anti-patterns detection approaches, event-B and ontology. Then Section 3 narrates the related works about anti-patterns detection. Section 4 introduces in details the proposed detection method. The method is applied in Section 5 while Section 6 analyzes the results. Finally, Section 7 presents the Conclusion.

2. Main Concepts

2.1. Static and Dynamic Analysis

Patterns can be checked using two types of program analysis, static and dynamic analysis. Static analysis is the analysis of program code or documentation. This type includes examining source code or abstract representations of source code. The advantage of examining source code is that no information is lost by compiling. Static analysis is the most common way to find the bad smells which are code taints such as long methods and code duplication. Bad smells are usually found using static analysis because those smells are merely present in the code. They are difficult to be detected at runtime because compilers usually remove information about code layout. Anti-Patterns are usually more structural problems, such as classes using an inappropriate hierarchy. Meta-models of the source code have the advantage that the detection approach is programming language independent "a dynamic analysis" or "design anti-pattern analysis". This type is usually achieved by injecting detection routines in the program binaries, so gathering information through dynamic analysis should always be done in an automated way.

2.2. Anti-patterns Detection Approaches

Current approaches for identification of anti-patterns operate either at the code level (for software re-engineering purposes) or at the design level (for design quality improvement purposes). Detecting anti-patterns at the design level allows the designer to anticipate the problems that could be generated by an implementation. Detection of anti-patterns at the code level is considered too late and may not reduce the correction cost. So, we are interested in anti-patterns detection at the design level. There are several metrics approaches that have been defined to detect anti-patterns as (Coupling, cohesion, complexity and inheritance).

In this paper, we will concentrate on the object-oriented design, where we propose a metric-based approach for anti-patterns detection on UML class diagram designs. The proposed method for detecting anti-patterns based on mathematical metric between UML model and other formal models.

2.3. Event-B

Event-B is a formal method for specifying, modeling and reasoning systems. Event-B is an evolution of the B-Method [4] developed by Jean-Raymond Abrial. A model in Event-B consists of contexts and machines, Contexts contain the static part (types and constants) of a model while Machines contain the dynamic part (variables and events). The model elements of a context [5], [6] are four types: sets, constants, axioms and theorems. Axioms are various predicates that describe the property of sets, constants and theorems. A context can extend to more than one context, and also can be seen by several machines. Clause "Theorems" lists the various theorems which have been proven within the context. A Machine consists of variables,

invariants, events, theorems and variants. Variables; define the state of a model. Invariants; constrain variables which are supposed to be held whenever variables are changed by an event. In Event-B, the state of a model is changed by means of event execution. Each event is composed of a name, a set of guards G (t; v) and some actions S (t; v), where t is the parameters of the event and v is the state of the system which is defined by variables. All events are Atomic and can be executed only when their guards hold [7].

There are various relationships between contexts and machines as illustrated in Fig. 1. A context can be "extended" by other contexts and "referenced" or "seen" by machines. A Machine can be "refined" by other machines and can reference to contexts as its static part.

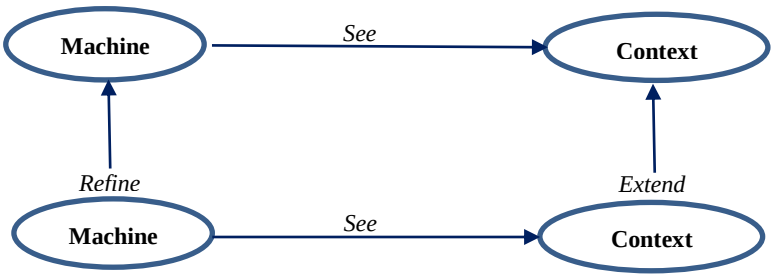

Fig. 1. Machine and context relationships.

Building a model usually starts with a very abstract model of the system, and then gradually details are added through several modeling steps in such a way that leads us towards a suitable implementation which called “refinement”. From a given model M1, a new model M2 can be built as a refinement of M1. In this case, model M1 is called an abstraction of M2, and model M2 will said to be a concrete version of M1. A concrete model is said to refine its abstraction [8].

2.4. Ontology

Ontologies are organized determinations of of concepts, properties, and relationships that are required for understanding the model. They includes customized knowledge for their specific environment of model[9]. Ontology is defined as a formal explicit specification of a shared conceptualization., where formal means that the ontology is machine readable, explicit means that the type of concepts should be defined and conceptualization refers to an abstract model for any phenomenon in the real world [10]. Ontology describes the concepts and the relationships that are important in a particular domain. That is to provide a vocabulary for the domain [11]. There are many tools to create and manipulate ontologies. Some tools are open source such as Protégé [12]. Ontology can perform a specific purpose as “Thesaurus” in the field of information retrieval or a model represented as OWL in the field of linked-data or XML (Extensible Markup Language) schema in the context of databases or in software development. There are many uses of ontologies as:

Helping in intelligent communication between human beings or organizations and software systems. Improving the quality of the design and software systems. Selecting semantic access and guided discovery of knowledge in Knowledge engineering and Management. Solving the problems that appear when we use different terminology to refer to the same concept or using the same term to refer to different concepts.

3. Related Work

This section summarizes earlier works in the anti-patterns detection methods.

Eman [7] proposed an approach to enhance the software quality and to detect the anti-patterns problems on the ATM model. The proposed approach improved the proof obligation by SMT solver. The Advantages of applying the proposed approach is not only integrating requirements, codes and verification in the system development life cycle, but also the consistence refinement, verification with high automation,

model re-usability, and detection anti-patterns problems. The proposed method in reference [7] was the base of the series of this work.

Fourati *et al* [2] proposed a metric based approach to detect certain five anti-patterns at code level, which affect the design level in UML class or sequence diagrams. They focused on applying the four metric methods on both diagrams. But they had the metric threshold issue like the others, so they are far from generalization.

Also, Alecsandar Stoianov [13] introduced a detection method for patterns and anti-patterns by a logic-based approach. The main advantage of that was the simplicity of the defining Prolog's predicates in describing both structural and behavioral aspects of patterns and anti-patters. But that was done only at code level.

Maria llano and Rob Pooley [14] contributed two folds. First, they defined the UML object oriented anti-patterns specifically for the good class. Then, they contributed transformations proposed for the correction of the anti-patterns manually. These transformations are based on local and structural refactoring. The authors of that work opened the door to detect anti-patterns in UML i.e. at design level.

Jaafar *et al* [15] presented an empirical study aimed at understanding the evolution of anti-patterns in 27 releases of three open-source software systems: ArgoUML, Mylyn, and Rhino. The results of that work showed that anti-patterns mutate from one type of anti-patterns to another, structural changes are behind these mutations. That led us to use standard conversion before detecting anti-patterns to avoid the mutations effects.

The authors in reference [16] used semi-formal structures to bridge the gap between requirements and Event-B models and keep the traceability to requirements in Event-B models, by using the benefits of the UML-B and Event Refinement Structures (ERS). But they didn't say anything about saving patterns and detection anti-patterns. This practical work gave us the idea of the standard converter in a certain phase of our proposed method.

Elasser [17] detected the anti-patterns in MOF and UML meta-models by using the QVTRelation (QVTR) transformation which distinguishes between two or more MOF-based models. A pattern is modeled with a Visual Pattern Modeling Language (VPML). This method detected the anti-Patterns through this transformation, but without any automatic suggestions to correct the anti-patterns. Although, this meta-model method have satisfactory performance but our proposed method gives automatically the number of appearances, the name of these anti-patterns, the way to correct these anti-patterns through OCL (OBJECT CONSTRAINT LANGUAGE) constraints, and also can detect the UML inconsistency.

Also, authors in reference [18] used an OntoUml editor to detect semantic anti-patterns in OWL but our proposed method used OntoUml editor to detect the semantic anti-patterns in UML class diagram.

Reference [19] presented the inconsistencies detection method in UML class diagram by using Net Beans 7.2.1 IDE. Our proposed method has a simple general way to detect the inconsistencies in addition to other anti-patterns.

4. Proposed Detection Method

This section is classified into two main subsections, one for coupling mathematical infrastructures UML class diagram with Ontology and event-B, while the other section is to present the proposed detection method.

4.1. Coupling Mathematical Infrastructures

On the basis of the mathematical infrastructure of UML, Ontology and event-B as formal methods, we will present the relation between them. Then we will suggest a new method to detect UML anti-patterns either structural or semantic in a class diagram type.

4.1.1. The relation between UML and ontology

Generally, we need to map UML to OWL ontology based for visual modeling and reusability.

In this subsection, we will present the relation between UML class diagram and OWL ontology based components. There exists a semantic correspondence between UML and OWL which allows the automatic translation or conversion from UML to OWL. Also Ontology and UML are both object oriented models, so they have similar meanings as in Table 1.

Table 1. Correspondence between UML and Ontolog

UML	Ontology
Package name	Namespace
Class	Class
Instance	Individual
Attributes	Properties
Multiplicity	Cardinality
Generalization	Subclass of (hierarchy)
Association	Relation
OCL constraints	Axioms containing these rules
(Superclass, subclass)	The hierarchy concept
FOL (FIRST ORDER LOGIC)	Axioms

4.1.2. The relation between UML and event B

This subsection presents the relation between UML and Event-B, where Classes are represented by B sets, constants, variables and operations and assembled into a single B component (i.e. machine or refinement, depending on the package stereotype).

Attributes are translated into variables whose type is a function from "the instances' set" to "the attribute type". The value of an attribute belongs to a particular instance can then be obtained by function application.

Associations are translated to functions in a manner similar to attributes except that the range of the function is based on the instances of the class at the supplier end of the association. Table 2 presents the

Table 2. Correspondence between UML B and Event B

UML-B	Event-B
Class(variable instances)	Variable $\subseteq$ Set
Class(fixed instances)	Set
Class (variable inst and has Super class)	Variable $\subseteq$ Super Class
Class(fixed inst and has super class)	Constant $\subseteq$ Super Class
Attribute (card 0..n-1..1)	Variable $\in$Class $\rightarrow$ Type
Attribute (card 0..n-0..1)	Variable $\in$Class $\nrightarrow$ Type
Attribute (card 0..n-0..n)	Variable $\in$Class $\leftrightarrow$ Type
Etc.(try other cardinalities in UML-B)	Etc.
Associations	As Attribute but Type is another class
Class Event	Event(self) WHEN self $\in$Class …
Class Constructor	Event(self) WHEN self $\in$SET\Class …
Class Invariant	$\forall$self·((self $\in$Class)$\Rightarrow$ Class invariant

4.2. The Proposed Method Phases

This subsection presents the four phases of the proposed method as shown in Fig. 2.

4.2.1. The converting phase

This phase relies on transformation UML class diagram to Event-B and OWL directly to save meta- model. In fact, the transformation of UML diagram to an XML document is handled by any UML editor. Note that, the XML mapping method loses the infrastructure information which we need to detect structural and semantic anti-patterns. Then we need to keep the UML meta-model or UML profile in our converter to handle the anti-patterns perfectly, so that is significant when using a direct converter.

In case of "Structural anti-pattern detection phase" the suitable converter is UMLB plugin in Rodin as event B editor. While in case of "Semantic anti-pattern detection phase" the suitable converters are semantic XML (SXML) by using SPARQL rules or XSLT processor to merge XML with UML profile OUP. Then the outputs of this phase are B model by UMLB and OWL ontology based by SXML.

4.2.2. The structural anti-pattern detection phase

This phase relies on UMLB model and Proof Obligation. That is by using any Event B editor to detect any structural anti-patterns on all UML class diagram components. UMLB tool generates event B model using translation rules. This step detects a group of anti-patterns. But when we apply proof obligation plugin, the method detects other group of anti-patterns.

4.2.3. The semantic anti-pattern detection phase

The structure detection of anti-patterns is insufficient so we need to cover the semantic information.

In this phase, we will try to detect all possible semantic anti-patterns by more than one way through making integration between five ways to get a pattern that is clear from any anti-pattern:

- Detecting anti-patterns through direct conversion from UML class diagram to OWL Ontology based.
- Detecting anti-patterns through merging the converted OWL ontology based of UML with WordNet as used in [2]. WordNet measures the similarity of meaning between two strings. But it's not enough. That is because there are semantic anti-patterns were not be detected by using WordNet or any linguistic ontology exactly in UML model.
- Detecting UML anti-patterns through using the validation of OWL anti-patterns in OntoUml editor which has a list of twenty one semantic anti-patterns giving us the name and the number of appearances. Some of them are explained in [18].
- Detecting anti-patterns through making Ontology verification syntactically in OntoUml which detect the anti-patterns.
- After OntoUml validation and ontology verification in the third and fourth steps, we will transform the OLED file to OWL file. That is to detect the "inconsistency anti-patterns" through " Reasoner" of Ontology, which will detect the anti-patterns as Similar name Error, Generalization and Disjointness.

Generally, according to this phase, the method detects thirty four different semantic anti-patterns. But some of anti-patterns can be detected by "Reasoner" and WordNet or by "Reasoner" and OntoUml. So Table 3 will list thirty four semantic anti-patterns.

Table 3. Semantic Anti-patterns Detection Ways

	The anti-pattern	The detection way
1	Class has no attributes	direct conversion
2	Class has no operations	direct conversion
3	Class has no attributes and no operations	direct conversion
4	Attribute has no multiplicity	direct conversion
5	Attribute has no Initial value	direct conversion
6	Operation has no return type	direct conversion
7	Association multiplicity is ambiguous	direct conversion
8	The same name error	Reasoner –WordNet
9	Generalization and Disjointness	Reasoner

11	Cyclic Inheritance	Reasoner –OntoUml
12	Relation Specialization (RS)	OntoUml
13	Relation Between Overlapping Subtypes (RBOS)	OntoUml
14	Mixin with same Rigidity	OntoUml
15	Binary relation with overlapping Ends	OntoUml
16	Deceiving Intersection	OntoUml
17	Relationally Dependent phase	OntoUml
18	Free role specialization	OntoUml
19	Generalization set with Mixed Rigidity	OntoUml
20	Heterogeneous Collective	OntoUml
21	Homogeneous Functional Complex	OntoUml
22	Imprecise Abstraction	OntoUml
23	Mixin with same Identity	OntoUml
24	Multiple relational Dependency	OntoUml
25	Part Composing overlapping whole	OntoUml
26	Relation composition	OntoUml
27	Relator Mediating Rigid Types	OntoUml
28	Repeatable relator Instances	OntoUml
29	Undefined Domain Formal Relation	OntoUml
30	Undefined phase partition	OntoUml
31	Whole Composed of Overlapping Parts	OntoUml
32	Attribute has no datatype	Ontology Verification
33	Class multiplicity equal zero	Ontology Verification
34	Invalid Stereotype	Ontology Verification

4.2.4. The correction phase

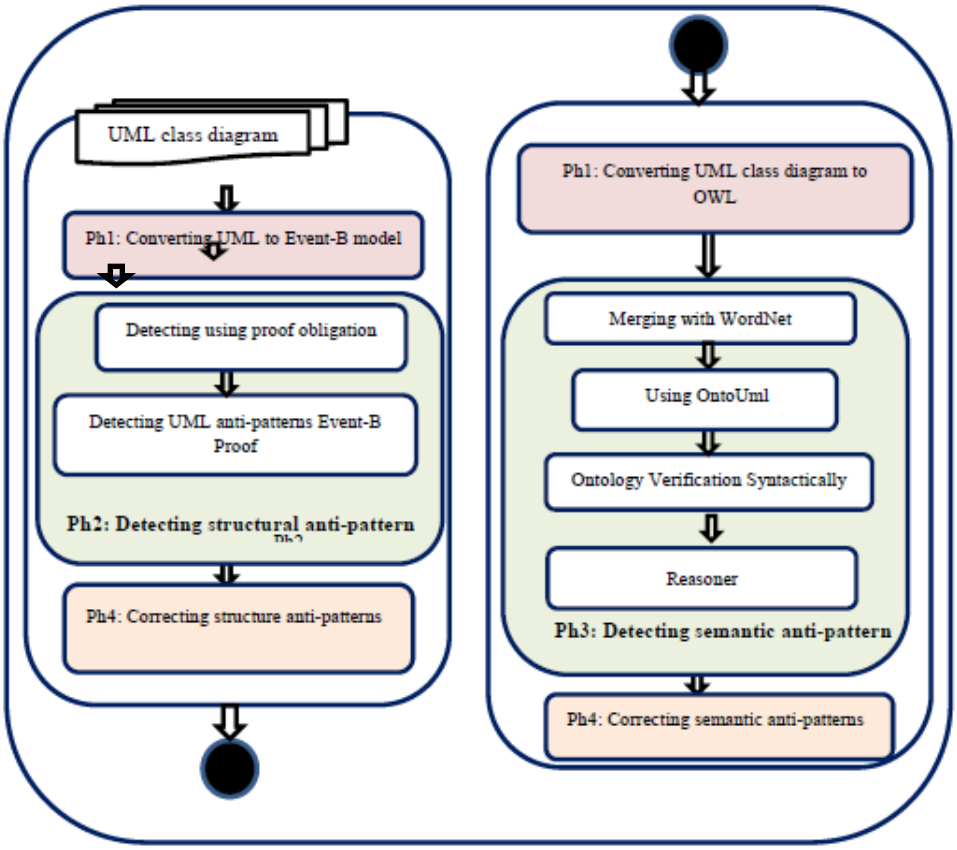

Fig. 2. The phases of the proposed method.

5. Case Study

This section presents the experimental case "Hospital UML class diagram" for applying the proposed method. That is by using the Rodin platform and its plugins [20] in the second phase and Protégé platform and its plugins [12] in the third phase. We apply the proposed method on a sample of nine UML class diagrams which were uploaded as UML templates to be used as patterns in references [21].

The Hospital UML class diagram contains six classes "Hospital, Booking, Doctor, Patient, Continuous and not Continuous ". Class "Booking" has five attributes and three operations, class "Doctor" has five attributes and no operations, class "Patient" has no attributes and four operations, class "Hospital" has two attributes and no operations, class "Continuous" has two attributes and no operations and finally, class "Not Continuous" has two attributes and no operations. The model also has seven associations, some of them with known multiplicity and some with unknown one. That is as shown in Fig. 3.

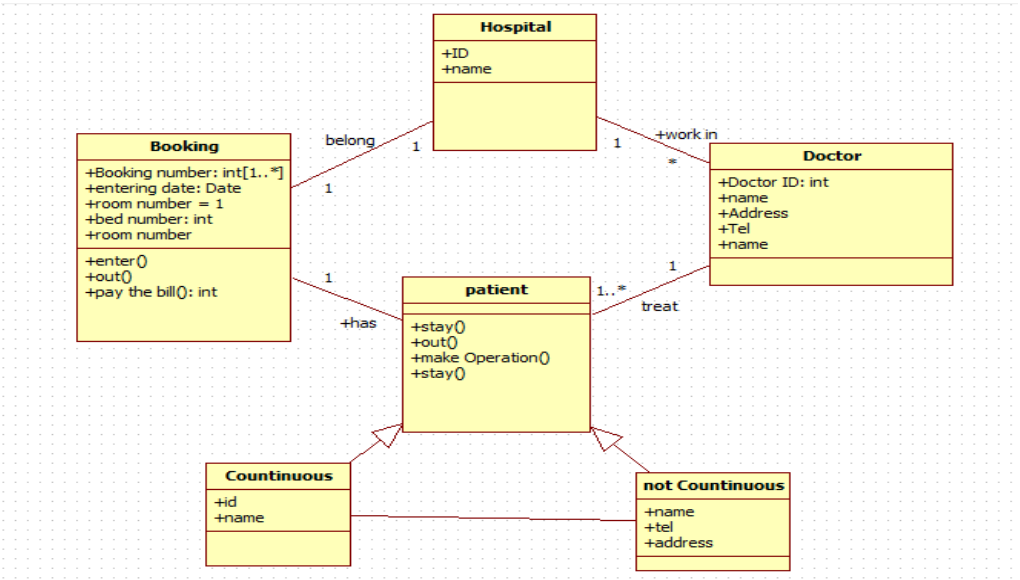

Fig. 3. Hospital UML class diagram.

5.1. The Converting Phase

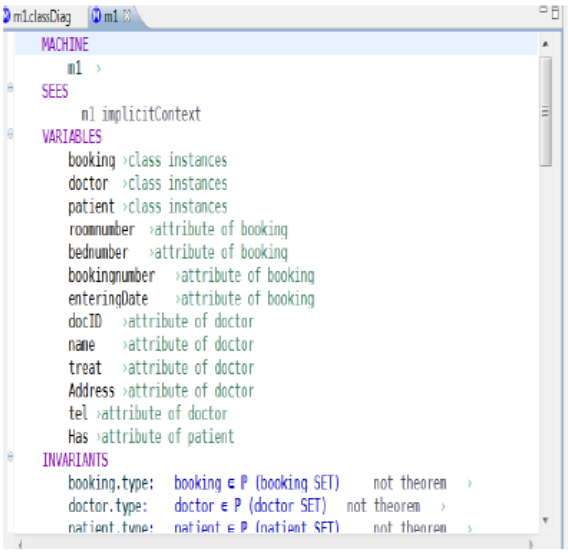

Fig. 4. Converting UML-B to event-b in rodin.

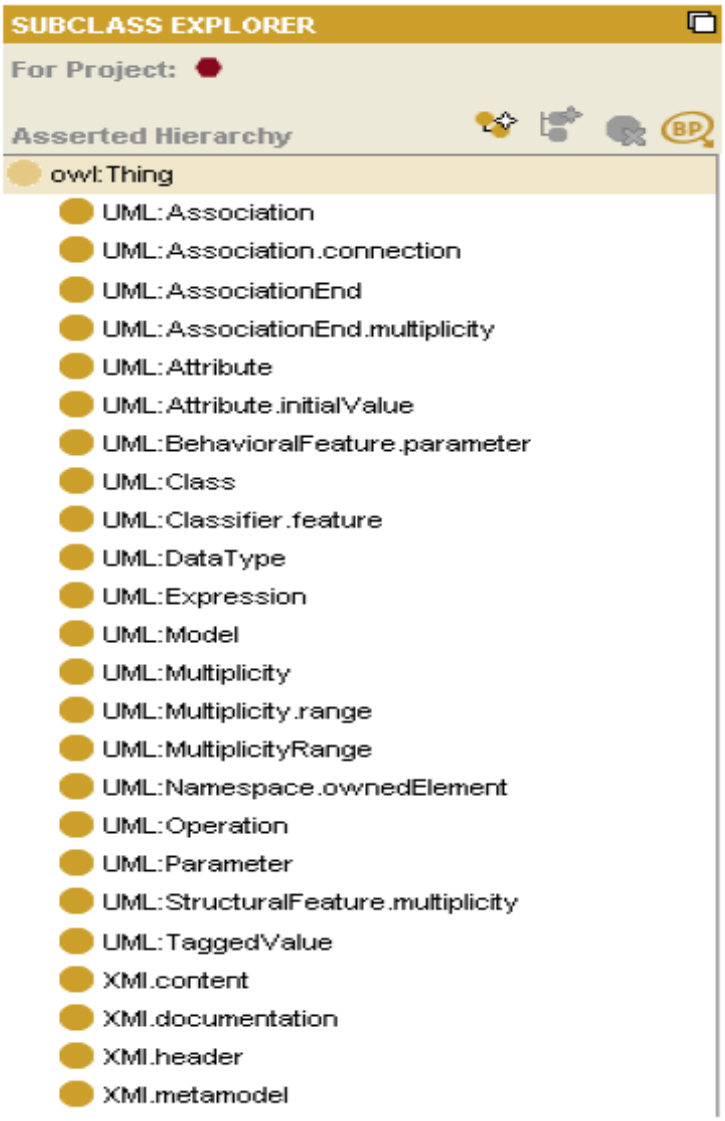

Fig. 5. Converting UML to OWL in Protégé.

The Fig. 4 and Fig. 5 present the screenshot for the converting phase results in Rodin for Event-B and Protégé for OWL ontology respectively.

5.2. The Structural Anti-pattern Detection Phase

The UML structural anti-patterns are detected practically in three levels:

- Some are detected during the first level in UML-B when we start using UML-B plugin.
- Others are detected during converting UML-B to Event B.
- Some are detected in “Proof Obligations”level.

The anti-patterns that have been detected in these levels are:

- The attributes of a class have no data type; attribute "Room number" in class "Booking" doesn't have data type. This is shown in Fig. 6.
- Operations have the same name; class "Patient" has two operations with the same name "Stay". This is shown in Fig. 7.
- The invalid identifier for association as shown in Fig. 8.
- The association name should not have any space where association "work in" has space in its name.

"The invalid expression constrain" anti-pattern is shown in the properties of the slot of "Patient" class which has Invalid expression guard of the event "Make operation". This is shown in Fig. 9.

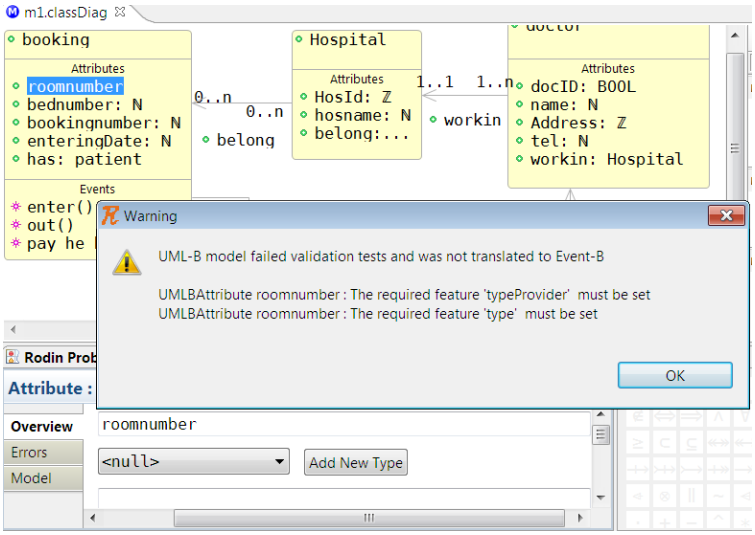

Fig. 6. An attribute doesn't have datatype.

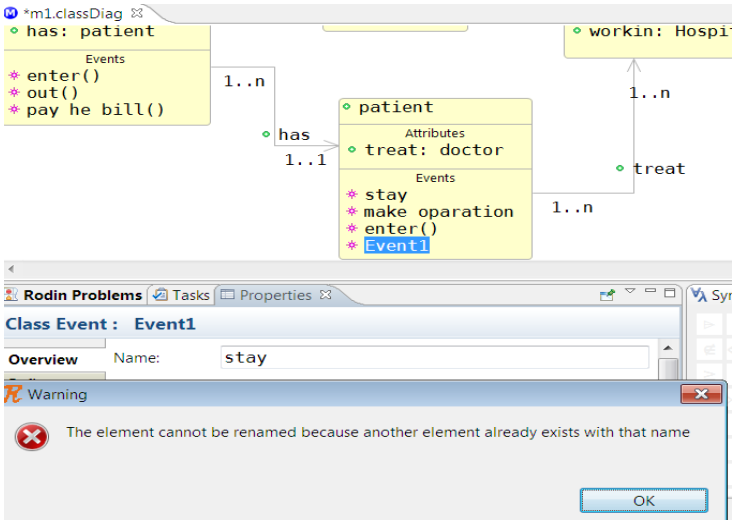

Fig. 7. Two operations have the same name.

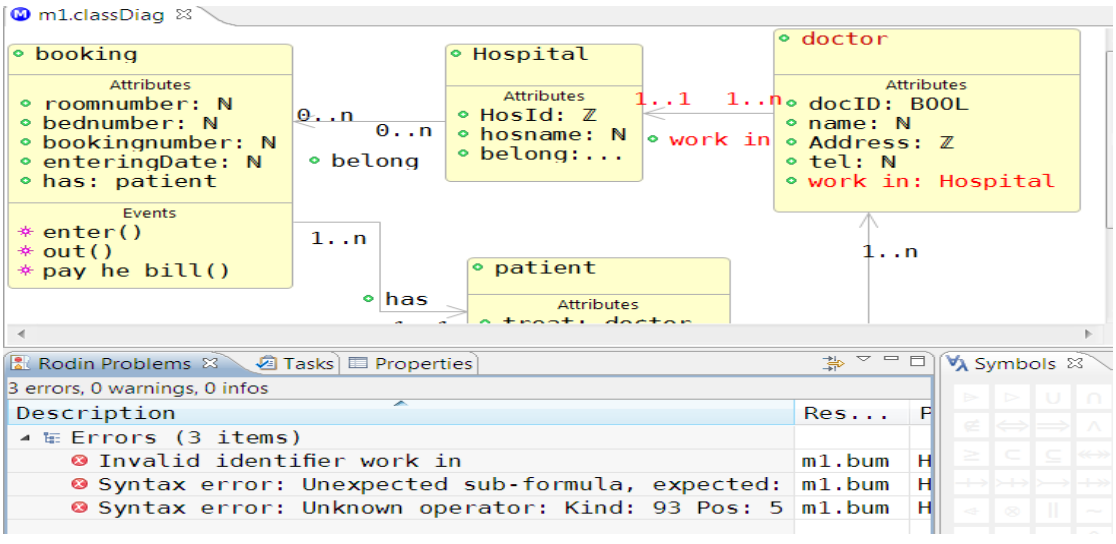

Fig. 8. Association "work in" has Invalid identifier.

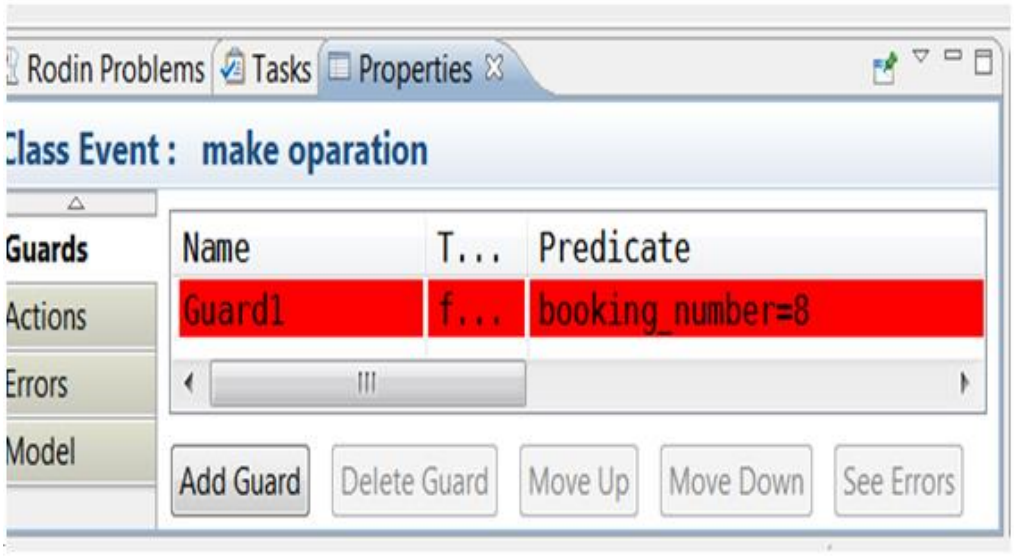

Fig. 9. Invalid expression guard of the event "make operation".

5.3. The Semantic Anti-patterns Detection Phase

This phase will detect semantic anti-patterns step by step as the following levels:

5.3.1. The direct conversion

When we convert UML class diagrams to an OWL ontology in Protégé (OWL editor) directly, it detects some anti-patterns as (Class has no operations, Class has no attributes, an attribute has no multiplicity, an attribute has no initial value, an operation has no return type and an association multiplicity is ambiguous). Classes "Doctor, Continuous and not Continuous" have no operations as shown in Fig. 10 and class "patient" has no attributes as shown in Fig. 11. Also, Fig. 12 shows the anti-pattern "The operation has no return type" where class "Booking" has operation "Pay the bill" with a return type "Integer", but operation "stay" in class "patient" has no return type , this anti-pattern is happening for all the rest of operations in class "Patient", and for operations "enter, out" in class "Booking". Finally, Fig. 13 shows the anti-pattern "association has no multiplicity", where there are seven associations, three of them have associations with known multiplicity, but the association connect classes "Booking" and "Patient" has two ends, one with name "Has" has no multiplicity and other with multiplicity.

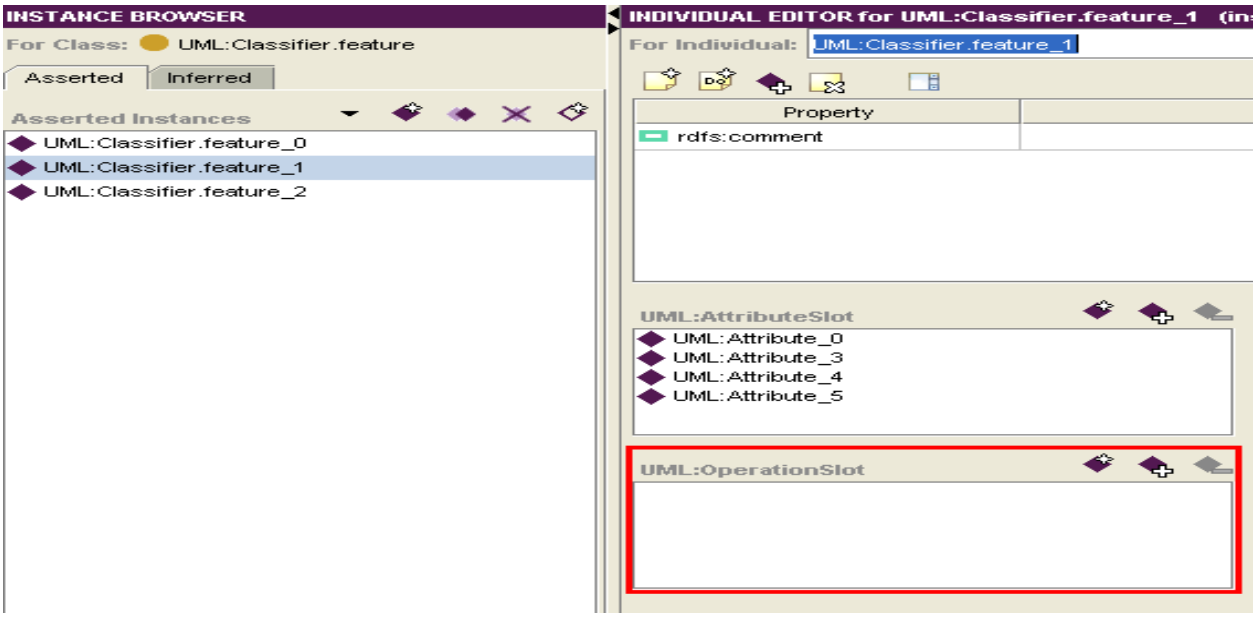

Fig. 10. Class (Doctor) has no operation.

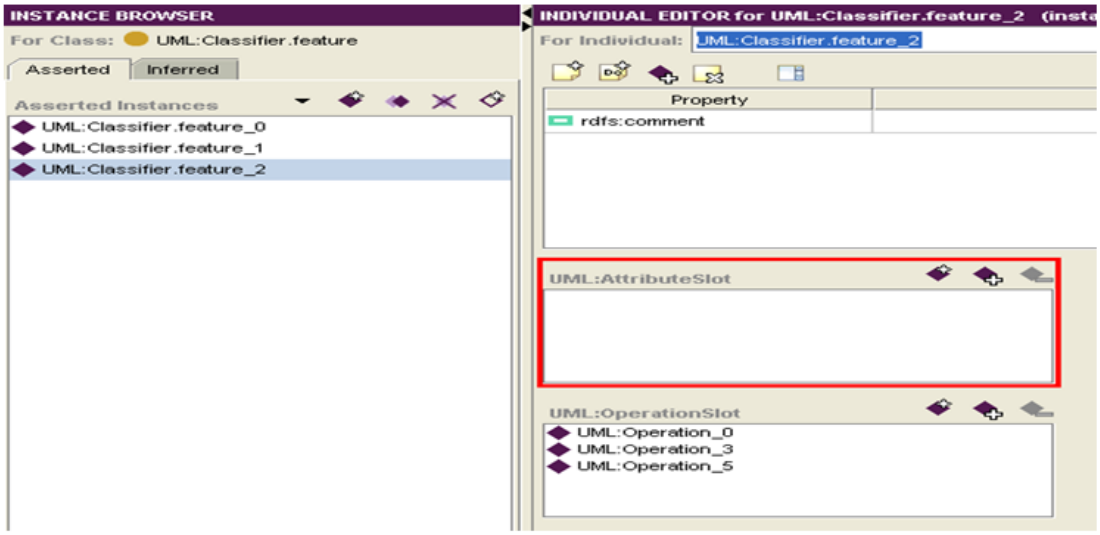

Fig. 11. Class (Patient) has no attributes.

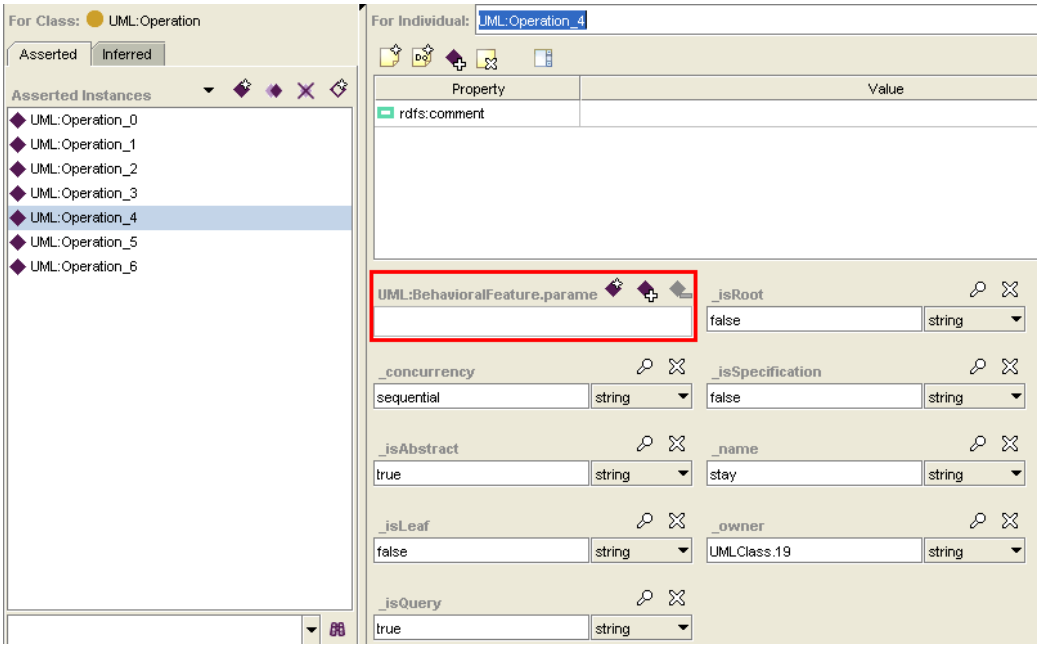

Fig. 12. An operation has no return type.

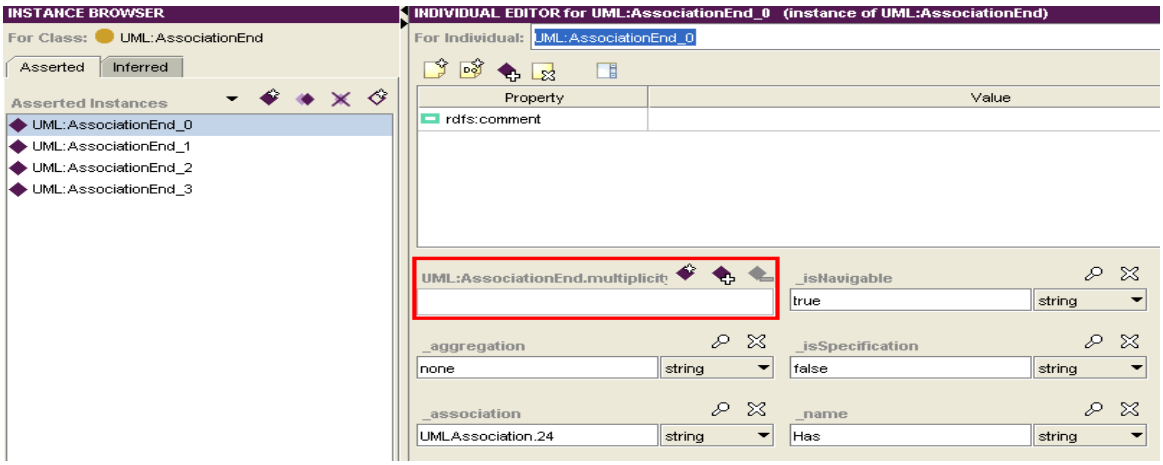

Fig. 13. An association has no multiplicity.

5.3.2. Merging UML ontology with WordNet

By merging UML Ontology with WordNet, the method detects the anti-pattern "Same name" at any components. In the Hospital UML class diagram, this anti-pattern was detected three times, class (Booking) has attributes with the same name "room number", class "Doctor" has attribute "name" repeated twice and finally, class "patient" has operation "stay" twice also.

5.3.3. Detection using OntoUml editor

OntoUml editor is an Ontology Lightweight Editor (OLED) [22]. It imports the EA (Enterprise Architecture) file which is created or imported from UML model. When UML model is inserted to OntoUml editor, it will detect the semantic anti-patterns which exist in the list of the 21 semantic anti-patterns.

In our example OntoUml editor detects three anti-patterns as shown in Fig. 14.

The anti-pattern "Association Cycle" where we have cyclic relation among classes "Hospital, Doctor, Patient and Booking".

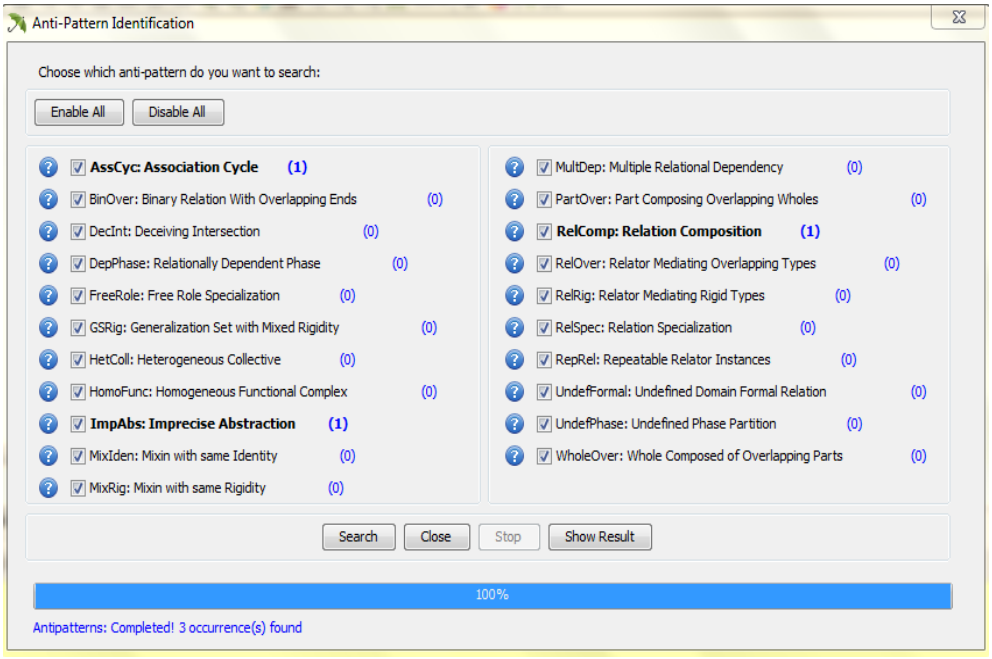

Fig. 14. OntoUml semantic anti-patterns detection.

The anti-pattern "Imprecise Abstraction" where we have class "Patient" with upper multiplicity greater than one has two subtypes and it has a relation with the class "Doctor".

The anti-pattern "Relation Composition" where there is an association between classes "Not Continuous " and " Continuous ".

5.3.4. Detecting using ontology verification syntactically

Type	Description	Stereotype	Element	Location
Syntactical	01. An attribute must have a minimum value of 1	Class	Hospital	EA_Model::<empty...
Application	02. Attribute type is null	Attribute	name	EA_Model::<empty...
Application	03. Attribute type is null	Attribute	name	EA_Model::<empty...
Application	04. Attribute type is null	Attribute	ID	EA_Model::<empty...

Fig. 15. Ontology verification detection.

OntoUml editor doesn't detect anti-patterns by validation only, but also can make Ontology verification

syntactically detect the anti-patterns (attribute has no data type, the class multiplicity equal zero and Invalid stereotype). In our example Hospital UML model, this verification detected the anti-patterns "Attribute has no data type" eight times, the anti-pattern "Class multiplicity equal zero "one time and "Invalid stereotype" nine times. This is shown in Fig. 15.

5.3.5. Detecting the inconsistency anti-patterns

According to reference [19], the class diagram inconsistencies are (Similar name, Generalization and Disjoint, Multiplicity constraints and Cyclic Inheritance). The transformation of OLED file to OWL file gives us the chance to detect the inconsistency using the "reasoner" plugin. In the Hospital UML model, "reasoner" plugin detected two anti-patterns "same name error" which are explained in session 5.3.2 and "Cyclic Inheritance" which explained in 5.3.3.

5.4. The Correction Phase

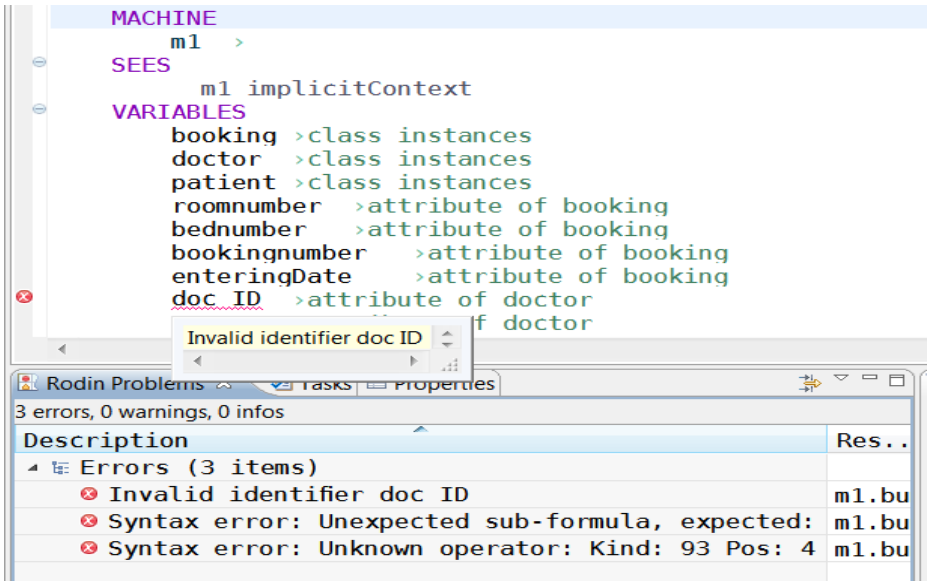

Fig. 16. Event B supports to solve anti-patterns.

This phase is pervasive in the previous phases where Event-B detects anti-patterns and also enables us to solve them as shown in Fig. 16. When we select any row from the problem list, we can know the location of the anti-pattern in Event B model. Also OWL ontology based gives the chance to make a refinement for any anti-pattern that is detected. We can describe that as a semiautomatic correction for design anti-patterns.

6. Analysis of Results

This section presents the analysis of the results when we apply the proposed method on a sample of nine UML class diagrams which is uploaded as UML templates to be used as patterns in several online references as in reference [21] and the "Hospital UML class diagram" model.

The proposed method detected seven structural anti-patterns which are (Attribute has no data type, Attributes have the same name, Attribute name has a space, Class name has a space, Association name is ambiguous, Associations have the same name and Association name has a space).

It also detected fourteen semantic anti-patterns which are (Class has no attributes, Class has no operations, Class has no attributes and operations, Operation has no return type, Attribute has no initial value, Attribute has no multiplicity, Association multiplicity is ambiguous, Attribute has no data type, Association cycle, Relation specialization, Invalid stereotype, Class multiplicity equal zero, Imprecise Abstraction and Relation Composition).

Table 4. Number of Appearances of the Detected Anti-patterns in Eight Groups

Anti-pattern Group	ATM	Customer Order	Seminar	Auction system	Library	HASP Java	Furniture system	Android system	CustomSWT widget	Hospital	# of appearance
Structure anti-pattern for Attribute	31	2	5	5	26	3	4		3	29	**108**
Structure anti-pattern for class	3	3		1	1					1	**9**
Structure anti-pattern for Association	14	16	24	9	21	2	6	9	11	12	**124**
Structure anti-pattern for operation										2	**2**
Semantic anti-pattern for Attributes					27		13			32	**72**
Semantic anti-pattern for class	7	18	13	10	17	14	14	35	16	9	**153**
Semantic anti-pattern for Association	3	2	4	2	11		1	3	2	5	**33**
Semantic anti-pattern for operation			1	2		3	1	5		6	**18**
Total	**58**	**41**	**47**	**29**	**103**	**22**	**39**	**52**	**32**	**96**	**519**

Generally, the proposed method detected twenty one anti-patterns which are grouped in eight groups; Structure anti-pattern for attribute, Structure anti-pattern for a class, Structure anti-pattern for association, Structure anti-pattern for operation, Semantic anti-pattern for attribute, Semantic anti-pattern for class, Semantic anti-pattern for the Association, and Semantic anti-pattern for operation. This is shown in Table 4 and Fig. 17. The "Semantic anti patterns for class" is the most commonly detected anti-pattern. Also the "Structure anti-patterns of association" appears too much, while the "Structure anti-patterns for operation" is the least commonly appeared anti-pattern. The total number of appeared anti-patterns in the sample of ten models is 519.

Table 5. The Precision and Recall for Each Group

Components	Attribute group	Class group	Association group	Operation group
Precision	75%	82%	80%	69%
Recall	81%	76%	75%	66%

Now to evaluate the proposed method accuracy, we calculate the precision and recall rates for each component as shown in Table 5. Generally, the proposed method precision is nearly 78% and the recall is 75%. These percentages are satisfactory for a method which detects both structural and semantic anti-patterns at design level. But we noted that the accuracy rate in case of "operation anti-pattern" group is the

minimum this come from the class diagram structure. Also the proposed method is more accurate for detecting class anti-pattern group than others.

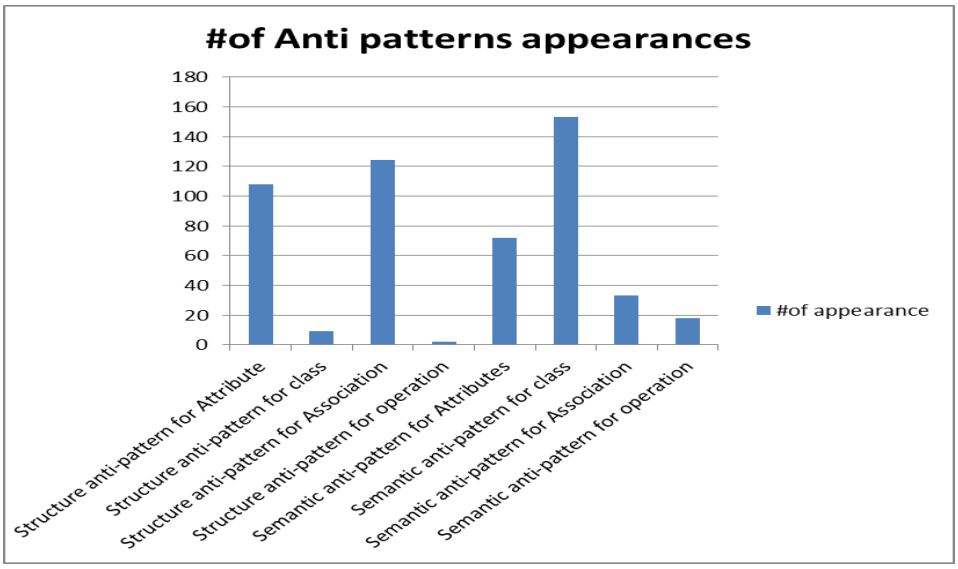

Fig. 17. Chart of number of appearances of design anti-patterns.

7. Conclusion and Future Work

The proposed method is used to evaluate the quality of UML patterns. That is to build a hard and stable base for new software projects. Definitely, there are no UML models without anti-patterns, so this paper proposed a general method for detection of anti-patterns. It scopes on the design for UML class diagram. The two main distinctions that differs our proposed method from others are; first is that it considers two types of anti-patterns structural and semantic anti-patterns which we grouped them into four groups of UML components; attributes, operations, classes, and association. Second, the correction phase is automatic in most of anti-patterns. Evaluations for several patterns were collected; our proposed method produced very satisfactory values of recall and precision.

In the future, we are going to make the anti-patterns automatically corrected. Also, the highest result will be produced when we create a plugin for OWL in Rodin or create a plugin for Event-B in Protégé. Finally, we will move to the next step by a general method that can detect both code and design anti-patterns levels.

References

[1] Stamelos, I. (2010). Software project management anti-patterns. *Journal of Systems and Software*, *83(1)*, 52-59.

[2] Fourati, R., Bouassida, N., & Abdallah, H. B. (2011). A metric-based approach for anti-pattern detection in uml designs. *Computer and Information Science*, 17-33. Berlin Heidelberg: Springer.

[3] Abdou, M., *et al*. (2012). SMURF: A SVM-based incremental anti-pattern detection approach. *Proceedings of the 19th Working Conference on Reverse Engineering (WCRE).*

[4] Abrial, J. R. (2005). *The B-book: Assigning Programs to Meanings.* Cambridge University Press.

[5] Boiten, E., & Abrial, J. R. (2012). Modeling in event-B-system and software engineering. *Journal of Functional Programming, 22(2)*, 217.

[6] Abrial, J. R., Butler, M., Hallerstede, S., & Voisin, L. (2006). An open extensible tool environment for Event-B. *Proceedings of International Conference on Formal Engineering Methods* (pp. 588-605). Berlin Heidelberg: Springer.

[7] Elsayed, E. K. (2014). Converting UML class diagram with anti-pattern problems into verified code relying on event-b. *AIML Journal, 14(1).*

[8] Fathabadi, A. S., Butler, M., & Rezazadeh, A. (2012). A systematic approach to atomicity decomposition in event-b. *Proceedings of International Conference on Software Engineering and Formal Methods* (pp. 78-93). Berlin Heidelberg: Springer.

[9] Yang, S., & Byun, H. (2016). Wide area ontology integration scheme for reasoning agents in surveillance networks. *Journal of Computers, 11(6),* 497-504.

[10] Corcho, O. (2004). *A Layered Approach to Ontology Translation with Knowledge Representation.* Doctoral dissertation, Informatica.

[11] Jiang, X., & Tan, A. H. (2009). Learning and inferencing in user ontology for personalized semantic web search. *Information Sciences, 179(16),* 2794-2808.

[12] Protégé Platform. Retrieved from http://protege.stanford.edu/

[13] Stoianov, A. (2010). Detecting patterns and antipatterns in software using prolog rules. *Proceedings of International Joint Conference on Computational Cybernetics and Technical Informatics (ICCC-CONTI)* (pp. 253-258).

[14] Llano, M. T., & Pooley, R. (2009). UML specification and correction of object-oriented anti-patterns, *Proceedings of 4th International Conference on Software Engineering Advances* (pp. 39-44).

[15] Jaafar, F., Khomh, F., Guéhéneuc, Y. G., & Zulkernine, M. (2014). Anti-pattern mutations and fault-proneness. *Proceedings of the 14th International Conference on Quality Software* (pp. 246-255).

[16] Alkhammash, E., Butler, M., Fathabadi, A. S., & Cîrstea, C. (2015). Building traceable event-B models from requirements. *Science of Computer Programming*, *111*, 318-338.

[17] Elaasar, M. E. (2012). An approach to design pattern and anti-pattern detection in mof-based modeling languages. Doctoral dissertation, Carleton University Ottawa.

[18] Guizzardi, G., & Sales, T. P. (2014). Detection, simulation and elimination of semantic anti-patterns in ontology-driven conceptual models. *Proceedings of International Conference on Conceptual Modeling* (pp. 363-376). Springer International Publishing.

[19] Idate, S. R., & Dalvi, N. I. (2014). Method to detect inconsistencies from class and sequence diagrams. *International Journal of Science, Engineering and Technology Research (IJSETR)*, *3(8).*

[20] Rodin platform. Retrieved from https://sourceforge.net/projects/Rodin-b-harp/

[21] Class diagrams. Retrieved from http://www.uml-diagrams.org

[22] OntoUML lightweight editor. Retrieved from https://code.google.com/p/ontouml-lightweight-editor/

Eman K. Elsayed received the Ph.D in computer science in 2005 at Alazhar University, and received the master of computer science at Cairo University in 1999, received the bachelor of science, mathematics and computer science at Department of Cairo University in 1994.

She published 32 papers until 2016 in data mining, ontology engineering, e-learning, image processing and software engineering. Also she published 2 books in formal methods and event B on Amazon database. She is a member in egyptian mathematical society and intelligent computer and information systems society.

Enas E. l-Sharawy received the computer science M.Sc in 2011 and Ph.D in 2014 from Al-Azhar University, She works as a lecturer in Computer Science Department in Al-Azhar University. She published many papers until 2017 in formal method, rodin platform and UML. She is interested in database and ontology.