Journal of King Saud University – Computer and Information Sciences (2015) 27, 170–180

King Saud University

Journal of King Saud University – Computer and Information Sciences

www.ksu.edu.sa
www.sciencedirect.com

Pattern-based model refactoring for the introduction association relationship

Boulbaba Ben Ammar *, Mohamed Tahar Bhiri

Faculty of Sciences of Sfax, Sfax University, Tunisia

Received 9 October 2013; revised 12 March 2014; accepted 5 June 2014
Available online 16 April 2015

KEYWORDS

Model refactoring;
UML;
B;
CSP;
Association relationship

Abstract Refactoring is an important software development process involving the restructuring of a model to improve its internal qualities without changing its external behavior. In this paper, we propose a new approach of model refactoring based on the combined use of UML, B and CSP. UML models are described by class diagrams, OCL constraints, and state machine diagrams. We detail a refactoring pattern that allows for the introduction of an association relationship between two existing classes. We illustrate our proposal by giving a case study involving the SAAT (Software Architecture Analysis Tool) system.

1. Introduction

Refactoring is a reorganization activity that aims to improve the internal structure of an existing body of code while maintaining its external behavior. This activity enhances the quality characteristics of a software system, including extensibility (during evolutionary maintenance), reusability, and efficiency. Various approaches have been proposed in the literature on the code refactoring technique. Fowler (1999) has, for instance, offered a catalog of refactoring rules applicable to the static part of a Java program, including "RenameClass", "ExtractClass", "MoveOperation", "MoveAttribute", and "RenameOperation".

More recently, the refactoring technique has also been adopted by several Agile software development methods (Shore and Warden, 2007) such as XP (Baumeister and Weber, 2013) and Scrum (Schwaber and Sutherland, 2013). In fact, they involve a Test-Driven Development (TDD) which is quick cycle consisting of three phases: test, coding and refactoring.

Refactoring tools are also available for most object-oriented languages, including Java, Smalltalk, C++, C#, Delphi and Eiffel, and for integrated development environments, such as Eclipse, NetBeans, and Oracle JDeveloper. These code refactoring rules have, however, often been defined informally, with no relationship being established between model quality and the rules. Several attempts have recently been made to overcome this inadequacy, with special focus on the application of the refactoring technique on standard models, including UML (Mens et al., 2007).

In this paper, we provide a new approach of model refactoring based on the combined use of UML, B (Abrial, 1996),

* Corresponding author.
E-mail addresses: Boulbaba.Ben-Ammar@live.fr (B. Ben Ammar), Tahar_Bhiri@yahoo.fr (M.T. Bhiri).
Peer review under responsibility of King Saud University.

ELSEVIER Production and hosting by Elsevier

http://dx.doi.org/10.1016/j.jksuci.2014.06.012
1319-1578

and CSP (Hoare, 2004). UML models are described by class diagrams, OCL constraints, and state machine diagrams. Specifically, we propose a catalog of refactoring patterns that are described in the same framework and formalized into B and CSP. These refactoring patterns cover the basic concepts of the object-oriented approach: conceptual relationships between classes (association and generalization/specialization), polymorphism, redefinition, abstract and generic class, and delegation.

The preservation of behavior after the application of refactoring is assigned to the tools associated to B (the prover of the ***Atelier B*** (Engineering, 2009)) and CSP (the model-checker ***FDR2*** (Goldsmith, 2005)). In fact, several researchers have defined systematic rules for the translation of UML into both B (Idani et al., 2009) and CSP (Rasch and Wehrheim, 2003) languages. Several studies have previously reported on the successful application of the B method in the development of various complex real-life applications, including the first driverless metro in the city of Paris, METEOR project (Behm et al., 1999). This method represents one of the few formal methods that has robust commercially available support tools for the entire development lifecycle, from specification down to code generation. Although this method is highly recommended for the verification of static properties such as safety, it is not used for checking dynamic properties such as liveness. For this reason, we have opted for the use of the CSP language.

The remaining parts of the paper will be structured as follows. Section 2 will provide an overview of related works on the topic under investigation. Section 3 will define our proposed approach. In Section 4, we will give a general description of the refactoring pattern. In Section 5, we will detail the pattern of association relationship introduction. Section 6 will be devoted to illustrating our proposal through the use of the SAAT system. Finally, the conclusion will summarize the major findings and provide new perspectives on model refactoring research.

2. Related works

Table 1 summarizes the major features characterizing the refactoring approaches so far proposed for the UML model using a set of evaluation criteria that are commonly cited in the literature. According to the MDE approach, refactoring can be considered a transaction processing system (Mens and Gorp, 2005) that introduces changes (without adding details) to the structure of a model. Unlike refinement, which is considered as a vertical model transformation, refactoring is a horizontal model transformation. In other words, the refactoring process does not lead to a change in the level of abstraction: the source model (before refactoring) and target model (after refactoring) remain in the same level of abstraction.

In software-driven engineering models, refactoring techniques are very limited (Allem and Mens, 2007). Several researchers (Gorp et al., 2003, Mens, 2006, Mens et al., 2007, Mens and Tourwe, 2004) indicate that taking the whole model refactoring process into consideration remains one of the challenging tasks. This process involves six major activities:

1. Identify which parts of the model should be refactored.
2. Decide on which refactoring rules to be applied to which areas.
3. Ensure that once applied refactoring would preserve model behavior and consistency.
4. Automate the application of refactoring.
5. Assess the impact of refactoring on software quality criteria (complexity, legibility, adaptability) or process (productivity, cost, effort).
6. Synchronize the refactored model and other artifacts, such as source code, documentation, specifications and tests.

The work described in Markovic (2008) offers a catalog of refactoring operations inspired by the list of operations previously described by Fowler (1999). The proposed operations are applicable on class diagrams and expressed by a QVT-based formalization of model transformation. The impacts of a refactoring operation on OCL constraints and object diagrams have also been described.

Other researchers (Gorp et al., 2003) proposed an extension of the UML meta-model that allowed for a better specification of two pre/postcontion operators in Refactoring: "Pull Up Method" and "Extract Method". This extension also conferred tools with other abilities: check pre/post-conditions,

Table 1 Summary of related works on the refactoring approaches for the UML model.

Approach of	Markovic (2008)	Mens (2006) and Mens and Gorp (2005)	van Kempen et al. (2005)	Mens et al. (2007)	Marković and Baar (2008)	Sunyé et al. (2001)	Correa and Werner (2007)
Consideration of class diagram	Yes	Partial	No	Yes	Yes	Yes	No
Consideration of state machine diagram	No	No	Yes	Yes	No	Yes	No
Consideration of OCL constraints	Yes	Yes	No	No	Yes	No	Yes
Behavior preservation	Transformation of model formalized into QVT	Meta-modeling	UML to CSP process	UML to graphs	Graph grammars	Rewriting	Rewriting
Tool	Supporting QVT	OCL query engine	Supporting CSP	Fujaba for the graph transformation	Formalism based on graph grammars	No	No
Detection of refactoring	No	Design smells	No	Best suited refactoring	No	No	OCL smells

compose sequences of refactoring operations, and use the OCL query engine to detect "design smells" (design flaws). In fact, The general idea is to provide basic rules of atomic transforming or refactoring. They can be treated as rewrite rules that may provide a basis for consistent restructuring. The potential occurrence of errors induced by refactoring activity would, therefore, be greatly reduced.

Marković and Baar (2008) also proposed a set of refactoring rules applicable to basic class diagrams considering OCL constraints. Their rules were inspired by the ones previously proposed in refactoring object-oriented languages (Mens and Tourwe, 2004). The authors defined the refactoring model as a set of transformation rules consisting of seven refactoring rules with or without effects on the syntax of the OCL constraints attached to the refactored class diagrams. The authors also used graph grammar formalism to check for behavior preservation.

The work described in Sunyé et al. (2001) has two lists of refactoring rules. The first list is applicable to the class diagram and has five basic operations: "addition", "removal", "move", "generalization", and "Specialization" of an element. It is worth noting here that an element can be a class, attribute, operation, or association end. The second list of rules is applicable to the state chart diagram and contains seven basic operations: "Unfold Exit Action", "Group States", "Fold Outgoing Transition", "Unfold Outgoing Transition", "Move into Composite State", "Move out of Composite State" and "Same Label". The semantics of these operations was defined in OCL.

In another study (Correa and Werner, 2007), the refactoring technique was used to improve the understanding and maintenance of OCL specifications. The authors identified the instances of poor OCL use (OCL smells) and offer a collection of adequate refactoring operations to prevent their occurrence. Among the OCL smells identified (a dozen) by the authors, we can quote: "Implies chain", "Redundancy", "Non-atomic rule," "And Chain", "ForAll chain" and "Long Journey".

Reimann et al. (2010, 2012) proposes a novel approach based on role models to specify generic refactorings, thus resolving the limitations of previous works and defining specific refactorings as extensions to generic ones. The approach was implemented based on the Eclipse Modeling Framework (EMF) and evaluated using multiple modeling languages and refactorings.

In Einarsson and Neukirchen (2012), the author has proposed automated refactorings that are developed for restructuring UML activity models together with their diagrammatic representation using the QVT operational transformation language for transforming UML models and diagrams created with the Papyrus UML editor.

The approaches mentioned above do not treat the UML models described simultaneously by class diagrams, state machine diagrams, and OCL constraints. This limitation does not help us to verify, after refactoring, the preservation of two essential properties, namely safety (class diagram and OCL constraints) and liveness (state machine diagram). Another problem of model refactoring is behavior preservation. By definition, a model refactoring is supposed to preserve the observable behavior of the model it transforms. In order to achieve this objective, we propose to translate a model to formal languages such as B and CSP.

2.1. Overview of UML basic concepts

UML defines four types of relationships between classes:

Association relationship: It describes a set of links or connections between classes. In Object Oriented Languages, this conceptual relationship is known as customer relationship.

Specialization/Generalization relationship: This is a conceptual relationship that allows a class, called subclass, to inherit the characteristics of its parent class, called superclass. In Object Oriented Languages, the specialization/generalization relationship is known as the inheritance relationship.

Realization relationship: It is a relationship in which an interface defines the contract guaranteed by an implementation class.

Dependency relationship: It does not necessarily require a link between the classes and shows that an element, or set of elements, requires other model elements for their specification or implementation. The dependency relationship is indicated by a dotted line pointing from the dependent (or client) to the independent (or supplier) elements.

In fact, UML defines several other concepts. We describe below some concepts that are of interest to our work.

Delegation: A class may delegate part of its activity to another class. In UML, the operation delegation mechanism is led by a relationship of composition or aggregation that connects the two classes.

Generic class: A class can have formal generic parameters representing types or variables. In UML, generic classes are called classes "template". We cannot use a template directly, we initially need the instancier. The instanciation implies via the dependence "bind" the binding of those formal generic parameters of the template to the real generic parameters, which gives a concrete class that can be used exactly like any ordinary class. Contrary to other programing languages such as Eiffel (Meyer, 2000), UML does not support the forced generics requiring the introduction of the heritage: the formal generic parameters representing types must go down from the ascending types.

Polymorphism: In the Object Oriented development process, a variable entity or an element of data structure can take several forms which, during execution, become attached with objects of the different types under the control of a static declaration.

2.2. Overview of the B method

Inspired by the works of Dijkstra (1975) and Hoare (2004), the B method was formulated and formally defined in the **B** Book (Abrial, 1996) of Jean-Raymond Abrial. It covers all the stages of software development, from specification to implementation, through the concept of refinement which allows for the rigorous expression of the properties required by specifications. The B method aims to produce a safe and accurate software construction. In fact, the B language is considered as an evolution of the Z language, aiming to develop a suitable approach for the whole development cycle as well as for practical industrial application. The B method distinguishes between two types of proofs: the conservation of the invariants and the correctness of refinement. The proofs of invariant conservation check model consistency and invariant properties,

which must be preserved at the initialization of and before/after the execution of the operations. The proofs of refinement correctness ensure the conformity of the concrete model as compared to its abstract counterpart. These proof obligations are automatically generated using the tools associated with the B method.

2.3. Overview of the CSP language

CSP processes are defined in terms of events considered as relevant for an object description. All the names of those events are called an alphabet. The simplest behavior of the process is to do nothing, a process which is denoted by STOP. To describe more elaborate behavior, CSP provides operators such as: prefix, recursion operators, deterministic and non-deterministic choice, hidden events, parallel composition, inputs/outputs, interleaving, and quantification. The three semantic domains (Roscoe, 1994) of CPS are stable traces, failures, and failures-divergences. A stable trace of a process behavior refers to a finite sequence of events that this process has been engaged in up to a given moment in time. The full set of all possible traces of a process `P` is denoted by `traces(P)`. The model then considers the stable failures associated with each process P which involves the couples **(t, E)**, where **t** refers to a finite set of traces accepted by `P` and `E` to the set of events that the process cannot run after running the events of `t`. All of those couples are denoted `failures(P)`. The CSP model also allows to characterize the deadlocks of `P`. In fact, if `E` is equal to the set of the executable events of `P`, then `P` is blocked. Finally, the model considers the set of failures-divergences associated with each process `P` which involves the set of all its potential failures and divergences. A process `P` is in a divergent state only if it is in a state where the only possible events are internal. The set of divergences of `P` denoted as `divergences(P)` is the set of traces `t` such that the process finds itself in a divergent state after running `t`. If the process is deterministic, then `divergences(P)` is empty.

2.4. Translation of the structural aspects of UML into B

Several approaches have been proposed for the translation of UML diagrams into B (Idani et al., 2009). Some studies (Ledang, 2001; Meyer and Souquières, 1999) suggested the development of an exhaustive approach for the simultaneous consideration of several UML diagrams. Other works (Laleau, 2002) focused on the area of databases to produce a safe SQL code. Still, other studies (Lano et al., 2004; Snook and Butler, 2004) proposed a B profile for UML utilization.

Table 2 Name assignment conventions.

$\mathcal{SM}$	The state machine diagram
s	Simple or composed state
$\mathcal{C}_s$	All direct successors of s
$\mathcal{T}_s$	The set of all couples (e, t) of successors t of s achieved by means of a transition triggered by e
$\mathcal{M}$	A sub-machine of $\mathcal{SM}$
$\mathcal{M}_{top}$	The state machine of high-level
$\mathcal{I}_{\mathcal{M}}$	The set of initial states of $\mathcal{M}$
$\varphi_{UML\rightarrow CSP}$	Translation function of state machine diagrams to CSP processes
SKIP	End of the CSP process
STOP	`deadlock`
$\|\|\|$	A parallel composition
;	A sequential composition

2.5. Translation of the behavioral aspects of UML into CSP

In this section, we describe the main aims of the work proposed by Rasch and Wehrheim (2003) for the translation of the state machine diagrams to CSP processes. Table 2 presents the name assignment conventions used in the remaining sections of the paper.

The translation function $\varphi_{UML\rightarrow CSP}$ of a state s is defined in CSP as follows:

$$\varphi_{UML\rightarrow CSP}(s) \equiv \begin{cases} \mathcal{P}_s = SKIP, \text{with } s \text{ is a final state,} \\ \mathcal{P}_s = \Box_{(e,t)\in\mathcal{T}_s} e \rightarrow \mathcal{P}_t, \text{with } s \text{ is a simple state and } \mathcal{C}_s = \emptyset, \\ \mathcal{P}_s = \sqcap_{t\in\mathcal{C}_s} \mathcal{P}_t, \text{ with } s \text{ is a simple state and } \mathcal{C}_s \neq \emptyset \text{or } s \text{ is an initial state,} \\ \mathcal{P}_s = (|||_{i=1}^{n} \mathcal{P}_{\mathcal{M}_i}); ((\sqcap_{t\in\mathcal{C}_s} \mathcal{P}_t) \not< \mathcal{C}_s \neq \emptyset \not> STOP), \text{ with } s \text{ is a composed state of sub-state machine } \mathcal{M}_i, \text{ with } 1 \leqslant i \leqslant n. \end{cases}$$

The translation function $\varphi_{UML\rightarrow CSP}$ of a sub-machine $\mathcal{M}$ is defined in CSP as follows:

$$\varphi_{UML\rightarrow CSP}(\mathcal{M}) \equiv \mathcal{P}_{\mathcal{M}} = \sqcap_{t\in\mathcal{I}_{\mathcal{M}}} \mathcal{P}_t$$

After calculating $\varphi_{UML\rightarrow CSP}(s)$ for any s and $\varphi_{UML\rightarrow CSP}(\mathcal{M})$ for any sub-machine $\mathcal{M}$ of $\mathcal{SM}$, the CSP process, corresponding to the state machine diagram $\mathcal{SM}$, can be calculated by combining the two obtained functions$\varphi_{UML\rightarrow CSP}$ and evaluating the state machine $\mathcal{M}_{top}$ as follows:

$$\mathcal{PROC}_{\mathcal{SM}} = \mathcal{M}_{top}$$

3. Proposed approach

Our approach consists in the development of a catalog of refactoring patterns that allow for the reorganization of the internal structure of UML/OCL class diagrams. These patterns can also transform the state machine diagrams while taking the modifications introduced to the class diagrams into account. Such patterns can be compared with the refactoring rules. The proposed refactoring patterns support the improvement of the software qualities, including extensibility, reusability, and efficiency, and allow model designers to introduce

concepts such as inheritance relationship, polymorphism, abstract and generic class, redefinition, association, and delegation. These patterns are characterized by a precise framework composed of four steps: identification of parameters, verification of applicability, evolution of specification, and correctness of pattern. These steps show the fundamental aspects of a refactoring pattern. Moreover, the proposed refactoring patterns are formalized in B, using systematic rules for transforming a class diagram and its OCL constraints into B (Ledang, 2001; Marcano and Levy, 2002; Meyer and Souquières, 1999) and the translation function $\varphi_{UML \to CSP}$ of a state machine diagram into the CSP process (Rasch and Wehrheim, 2003). This helps us to identify precisely the different conditions of application, the evolution of a UML class diagram, the OCL constraints and state machine diagram, and the correctness of the pattern. The various checks are entrusted to the ***AtelierB*** for B specifications and ***FDR2*** for the CSP process. These proposals can serve as milestones for the eventual automation of refactoring patterns.

The main idea of our work is the proposal of a catalog of refactoring patterns allowing the addition of the various relations and concepts described above, with the aim of improving the quality factors of an existing UML specification. Certain patterns enhance the addition of new relationships between non-dependent classes, which expresses the need for introducing concepts into a given class. The refactoring patterns associated with inheritance, association, redefinition, and polymorphism are parameterized by the following:

- the two involved classes,
- OCL constraints attached to each class,
- state machine diagrams corresponding to the dynamic properties of each class instance.

While refactoring patterns are associated with the delegation Ben Ammar et al. (2008), the generic and abstract classes are set by the following:

- the relevant class,
- OCL constraints attached to the relevant class,
- the state machine diagram corresponding to the relevant class.

4. Description of a refactoring pattern

The application of a refactoring pattern involves four steps. We first identify its parameters. We then ensure that these parameters satisfy a set of conditions. After that, we show the different updates provided by the selected pattern. We finally check the results produced. The definition of a refactoring pattern is, therefore, composed of four steps that can be summarized as follows:

1. Parameter identification
2. Applicability verification
3. Specification evolution
4. Pattern correctness

4.1. Parameter identification

It presents the pattern parameters that take the form of one or two triplet(s) containing the relevant class, the OCL constraints attached to the relevant class, and the state machine diagram associated to the relevant class: Class, OCL_Class, STD_Class. with:

- *Class* represents the involved class.It is characterized by a set of static properties, called *l_attr_Class*, and a set of dynamic properties, called *l_meth_Class*.
- *OCL_Class* represents the OCL constraints. These constraints, attached to *Class*, consist of the following elements:
  - *I_Class*: the invariant of Class, a condition that must be checked for all the objects of the class in every stable moment,[1] is formalized in OCL by:

    ```
    Context Class
    invI_Class : condition
    ```

  - for each method *meth_Class* of *l_meth_Class*:

    P_meth_Class: the precondition of the operation *meth_Class*, a condition which must be checked before the execution of *meth_Class*,
    Q_meth_Class: the postcondition of the operation *meth_Class*, a condition which must be checked after the execution of *meth_Class*, formalized in OCL by:

    ```
    Context Class :: meth_Class()
    pre P_meth_Class : condition
    post Q_meth_Class : condition
    ```

- *STD_Class* constitutes the state machine diagram which models the behavior of *Class* instances.

4.2. Applicability verification

A refactoring pattern is applied with the identified parameters. It is advisable to ensure that these parameters satisfy the two following conditions:

1. Coherence between the various parameters of the pattern which concerns, respectively:
   - `Class` **and** `OCL_Class` by checking the adequacy of the OCL constraints for the properties of the class.
   - `Class` **and** `STD_Class` by the possibility of carrying out the methods called upon in sequence according to `STD_Class`.
2. Consistency of the concept introduced by the pattern (inheritance, association, delegation,etc.).

The enumerated conditions can be classified into two types:

- those pertaining to static properties,
- those related to dynamic properties.

To check the conditions related to the static properties (safety properties), we transform the classes and their properties and OCL constraints into a B specification using the following systematic translation rules:

- from UML into **B** as proposed in Ledang (2001) and Meyer and Souquières (1999),
- from OCL into **B** as proposed in Ledang and Souquières (2002) and Marcano and Levy (2002).

[1] A "stable moment" corresponds to the moment which follows the execution of a method of the class. During, the execution of a method, the invariant of class can be temporarily violated.

The tool associated with B is ***Atelier B*** (Engineering, 2009).

To check the conditions pertaining to dynamic properties (liveness properties), we transform the state machine diagrams into CSP processes using the function $\varphi_{UML \to CSP}$ proposed by Rasch and Wehrheim (2003). The tool for checking the CSP processes is ***FDR2*** (Goldsmith, 2005).

The recourses to the translation into B and CSP are justified by the non-availability of tools for directly checking UML/OCL specifications. In this respect, several proposals involving the translation of UML/OCL specifications towards formal languages and allowing for the use of the associated tools are available in the literature.

The process of verifying parameter coherence is as follows:

- translate `Class` into a B machine, called B_Class,
- translate `OCL_Class` into an invariant expression in the corresponding machine,
- add control annotations, proposed by Ifill (Ifill et al., 2007), in the `ASSERTIONS` clause of the B machine. For each method $meth_i$*_Class*, an assertion contains two proof obligations in the following form:

$$I\_Class \wedge P\_meth_i\_Class \Rightarrow [Q\_meth_i\_Class](P\_meth_j\_Class)$$
$$I\_Class \wedge P\_meth_i\_Class \Rightarrow [Q\_meth_i\_Class](P\_meth_k\_Class)^{a}$$

[a] $meth_j$*_Class* and $meth_K$*_Class* can be called after the execution of $meth_i$*_Class*

The coherence of the B machine obtained proves the coherence between *Class* and its constraints *OCL_Class*.

The validation of the assertions proves the coherence between *Class* and its state machine diagram *STD_Class*.

The verification process of the consistency of the concept introduced by the pattern will be described for each pattern as the analysis unfolds.

4.3. Specification evolution

It presents the various updates automatically realized for the UML/OCL specifications.

4.4. Pattern correctness

It concerns:

1. The safeguarding of the properties of the restructured classes.
2. The preservation of the behaviors of the restructured classes.

We describe below only the definition of the refactoring pattern: Introduction of the concept of association.

5. Definition of pattern

The association relationship is a semantic connection between two or more classes. It can be binary or n-ary. An association is characterized by the following:

- its multiplicity is used to specify the minimum and maximum number of instances of each class in the relation between two or more classes,
- its navigability shows how to access from one class into another. If the relationship is between *Class*1 and *Class*2 and only *Class*2 is navigable, then you could access to *Class*2 from Class1 but not inversely. This means that while navigability is, by default, bidirectional, association is mono-directional.

In UML, a class can use one or more static (attributes) and dynamic (methods) properties of one or more other classes. Accordingly, the class must have an association relationship with the classes consulted, with a navigability towards those classes.

In this section, we limit ourselves to the introduction of a binary mono-directional association at the time involving the call of one or more methods.

5.1. Parameter identification

The refactoring pattern of the introduction of the association concept is parameterized by:

$\langle Class1, OCL\_Class1, STD\_Class1 \rangle$
$\langle Class2, OCL\_Class2, STD\_Class2 \rangle$

Hypothesis: The associations between classes indicate method invocations, i.e., if *Class*1 has an association with *Class*3, then *Class*1 calls at least a method of *Class*3 (see Fig. 2).

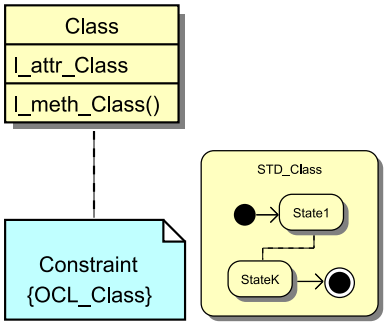

Figure 1 Parameters before refactoring.

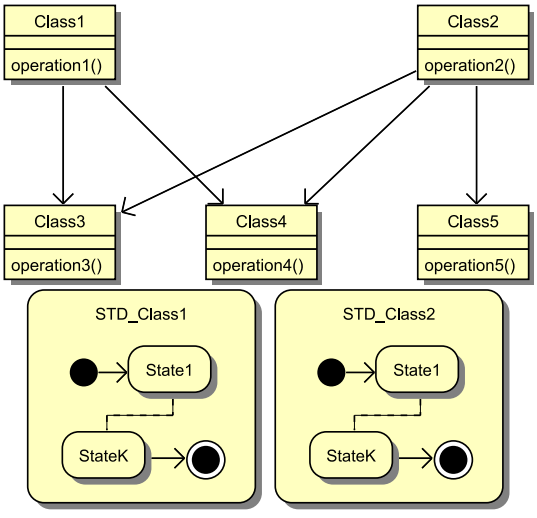

Figure 2 Parameters before refactoring.

5.2. Applicability verification

In order to concretize our pattern, we suppose that the behavior of `Class2` is partly carried out by `Class1`. Thus, we suppose that `Class2` delegates its work to the connected classes in the following way: `Class3` executes `operation3`, then `Class4` executes `operation4`, and finally `Class5` executes `operation5`. Similarly, `Class1` delegates its work to the connected classes in the following way: `Class3` executes `operation3`, and then `Class4` executes `operation4`.

The behavior of a class instances is modeled by a state machine diagram, and the detection of the invoked methods is verified by CSP processes.

The behaviors of `Class1` and `Class2` are defined by the processes `PSM_Class1` and `PSM_Class2`, respectively.

$$PSM\_Class1 = Class3!operation3 \rightarrow Class3\_r?x \rightarrow$$
$$Class4!operation4 \rightarrow Class4\_r?x \rightarrow STOP$$
$$PSM\_Class2 = Class3!operation3 \rightarrow Class3\_r?x \rightarrow$$
$$Class4!operation4 \rightarrow Class4\_r?x$$
$$\rightarrow Class5!operation5 \rightarrow Class5\_r?x \rightarrow SKIP$$

Accordingly, to check the existence of a method call, we must find a textual substitution for the described CSP processes.

5.3. Specification evolution

The evolution of the specification consists of:

- the introduction of an association between `Class1` and `Class2`, with a navigability from `Class2` to `Class1`.
- suppression of associations between `Class2` and `Class3`, and between `Class2` and `Class4`.

Fig. 3 presents the state of specification after refactoring.

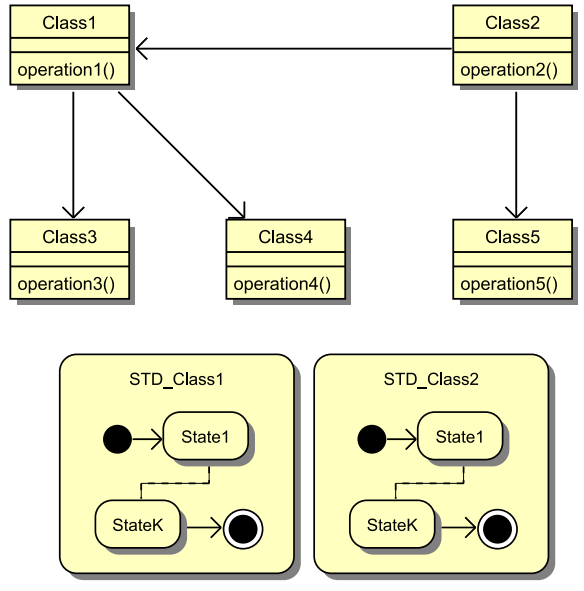

Figure 3 Parameters after refactoring.

5.4. Pattern correctness

The changes made by the application of the proposed refactoring pattern preserve the properties of the two classes, particularly that the static and dynamic aspects of `Class1` and `Class2` have not been changed. In fact, the direct relationship between `Class2` and `Class3` is replaced by the two relations between `Class2` and `Class1` followed by `Class1` and `Class3`. The verification of the B specification of the chained decomposition of an association is described in Ben Ammar (2012).

Changes to state machine diagrams `STD_Class1` and `STD_Class2` require the verification of whether the behavior of those two diagrams in relation to their correspondent abstract levels is preserved.

Accordingly, it suffices to show that:

$$assertPSM\_Class1\_r \sqsubseteq_{\tau} PSM\_Class1$$
$$assertPSM\_Class2\_r \sqsubseteq_{\tau} PSM\_Class2$$

with $\sqsubseteq_{\tau}$ referring to the CSP refinement based on the model of the traces.

6. Case study

To illustrate our proposal, we take the class diagram of SAAT (Software Structures Analysis Tool) system (van Kempen et al., 2005) as an example. The SAAT is an analytical tool used to calculate the parameters in a UML model, which can then be used to analyze the potential model or the defects of anti-patterns.

Classes: The class diagram (see Fig. 4), corresponding to the SAAT system, consists of the following classes: *Saat*, *DB*, *Stat*, *DBCreate*, *Parser*, *DBFill*, *DBCheck*, *Analyse*, *StatCalc*, and *StatFilter*. The associations between those classes indicate the invocations of methods, i.e. if a class *A* has an association with the class *B*, then class *A* calls a method of the class *B*.

Fig. 4 presents the initial class diagram corresponding to the SAAT system. The class *Saat* delegates its work to the associated classes in a sequential way: the database is created (*DBCreate.create*()), an input file is analyzed (*Parser.parse*()), and the data are inserted in the database (*DBFill.fill*()). After the insertion of the data, the filled database is checked (*DBCheck.check*()).

After that, the data will be analyzed (*Analyse.analyse*()), and the statistics are calculated (*StatCalc.calculate*()) and filtered

(*StatFilter.filter*()) according to the criteria defined by the user.

OCL constraints: There are no important OCL constraints attached to the various classes.

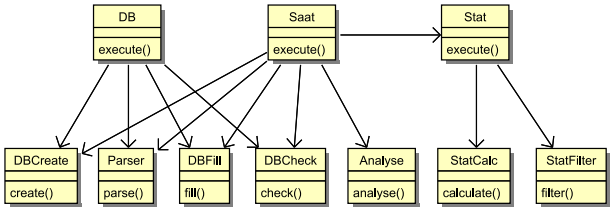

Figure 4 Class diagram before refactoring.

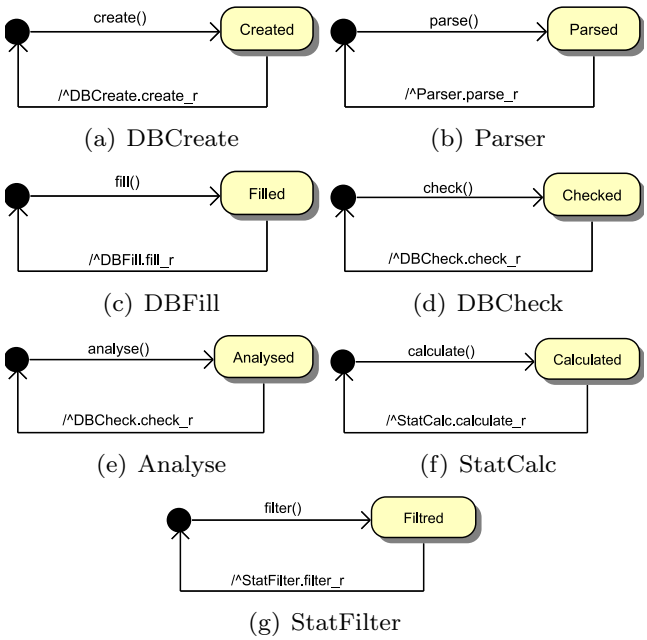

Figure 5 State machine diagrams of: DBCreate, Parser, DBFill, DBCheck, Analyse, StatCalc & StatFilter.

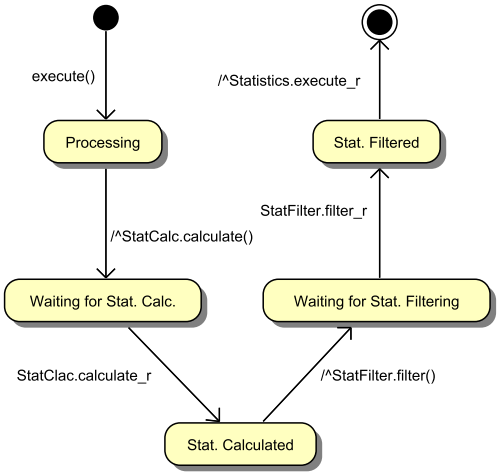

Figure 6 State machine diagram of Statistics.

State machine Diagrams: The behaviors corresponding to the various class instances are presented in Figs. 5–8.

The diagrams presented in Figs. 4–8 show a situation where one can apply the refactoring pattern: introduction of the association relationship between *Saat* and DB.

6.1. Parameter identification

The parameters of the pattern are:

$\langle Saat, OCL\_Saat, STD\_Saat \rangle$

$\langle DB, OCL\_DB, STD\_DB \rangle$

6.2. Applicability verification

In the following, we describe the CSP processes corresponding to the systematic transformations of the state machine diagrams from all the classes of the SAAT system.

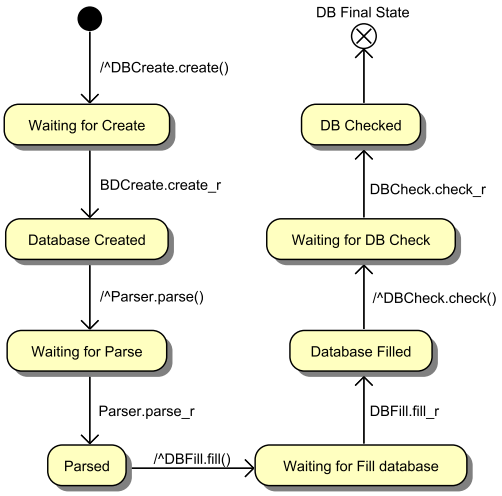

Figure 7 State machine diagram of DB before refactoring.

```
PSM_DBCreate = DBCreate?x → create() →
DBCreate_r!create → PSM_DBCreate
PSM_Parser = Parser?x → parse() →
Parser_r!parse → PSM_Parser
PSM_DBFill = DBFill?x → fill() →
DBFill_r!fill → PSM_DBFill
PSM_DBCheck = DBCheck?x → check() →
DBCheck_r!check → PSM_DBCheck
PSM_Analyse = Analyse?x → analyse() →
Analyse_r!analyse → PSM_Analyse
PSM_StatCalc = StatCalc?x → calculate() →
StatCalc_r!calculate → PSM_StatCalc
PSM_StatFilter = StatFilter?x → filter() →
StatFilter_r!filter → PSM_StatFilter
PSM_Saat = (DBCreate!create → DBCreate_r?x →
Parser!parse → Parser_r?x → DBFill!fill →
DBFill_r?x → DBCheck!check → DBCheck_r?x) →
Analyse!analyse → Analyse_r?x → Stat!execute →
Stat_r?x → SKIP
PSM_DB = (DBCreate!create → DBCreate_r?x →
Parser!parse → Parser_r?x → DBFill!fill →
DBFill_r?x →DBCheck!check → DBCheck_r?x) → STOP
PSM_Stat = Stat?x → StatCalc!calculate →
StatCalc_r?x → StatFilter!filter →
StatFilter_r?x → Stat
```

From these CSP processes, we deduce that the PSM_Saat process requires the execution of the method calling sequence of the process PSM_DB. Hence, an association relationship between these two classes proves to be necessary.

6.3. Specification evolution

The application of the refactoring pattern on the class diagram presented in Fig. 4 generates the class diagram proposed in Fig. 9.

The refactoring pattern proposed in this work was noted to improve the architecture of the SAAT application. In fact, the class diagram was noted to display less links after refactoring (9 association relationships) than before refactoring (12). This would facilitate extensibility of the SAAT application.

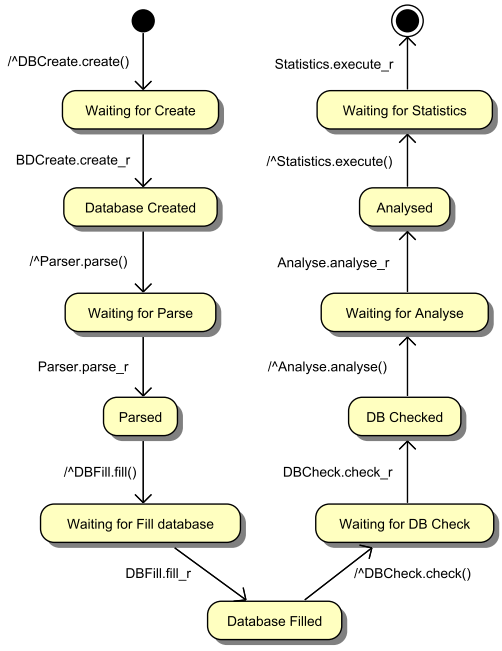

Figure 8 State machine diagram of Saat before refactoring.

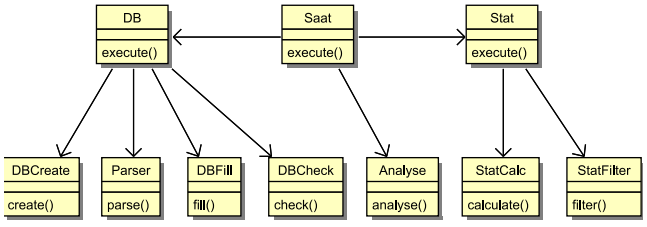

Figure 9 Class diagram after refactoring.

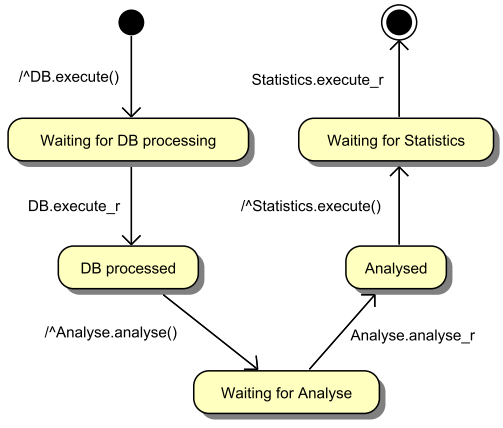

Figure 10 State machine diagram of Saat after refactoring.

Furthermore, the modifications on the level of state machine diagrams of `Saat` and `DB` are presented in Figs. 10 and 11.

Moreover, the state machine diagram of `Saat` generated after refactoring was noted to be less bulky than the one displayed prior to refactoring. This promote would enhance the comprehension of the behavioral aspects of the SAAT Application. The state machine diagrams corresponding to the other classes remain unchanged.

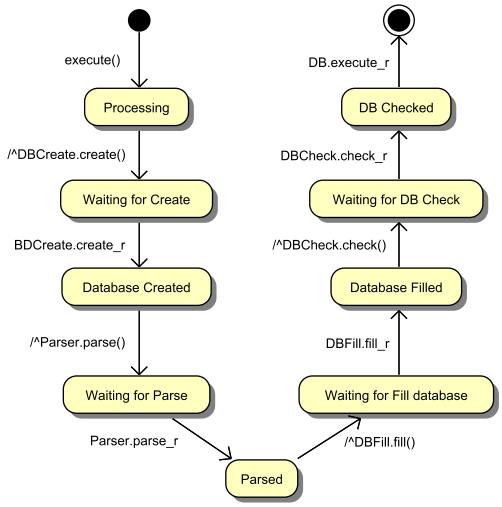

Figure 11 State machine diagram of DB after refactoring.

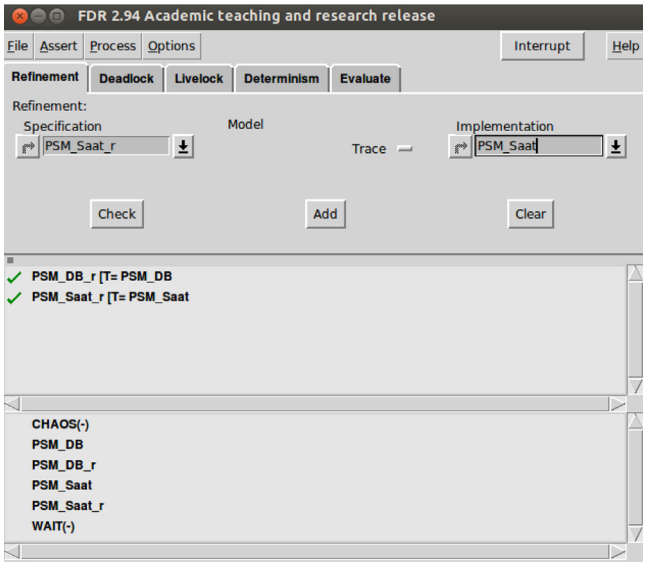

Figure 12 Verification with ***FDR2***.

6.4. Pattern correctness

After refactoring, we have to check the behavior preservation of the SAAT system. Accordingly, we observe the verification rules described in the previous section.

```
PSM_Saat_r = DB!execute → DB_r?x →
Analyse!analyse
→ Analyse_r?x → Stat!execute → Stat_r?x → SKIP
PSM_DB_r = DB?x → DBCreate!create → DBCreate_r?x →
Parser!parse → Parser_r?x → DBFill!fill →
DBFill_r?x
→ DBCheck!check → DBCheck_r?x → DB_r!execute → DB
```

Using ***FDR2***[2], we easily prove the consistency of the association relationship to be added between `Saat` and `DB`, expressed by the two following assertions:

`assert PSM_DB_r` $\sqsubseteq_{\tau}$ `PSM_DB`

`assert PSM_Saat_r` $\sqsubseteq_{\tau}$ `PSM_Saat`

These assertions are checked by ***FDR2*** (see Fig. 12).

7. Conclusion

Refactoring is a well-known technique for the enhancement of software quality, particularly in terms of extensibility, reutilisability, and effeciency. It is frequently applied to the code. The central problem of the refactoring technique lies in behavior preservation following the execution of the reorganization process. This paper pleads in favor of applying the refactoring technique at an advanced stage of software development. The refactoring pattern for introducing the association relationship proposed in this paper is applied to class diagrams, OCL constraints, and state machine diagrams to obtain high quality UML models, i.e. correct, extensible, reusable and effective UML models. In this respect, the authors (Ben Ammar, 2012), have previously proposed other refactoring patterns that allow:

1. Introduction of the concept of inheritance.
2. Introduction of the concept of redefinition.
3. Introduction of the concept of abstract class.
4. Introduction of the concept of polymorphism.
5. Introduction of the concept of delegation (Ben Ammar et al., 2008).
6. Introduction of the concept of generic class.

The list of refactoring patterns mentioned above is elementary. It covers the introduction of the concepts of inheritance, association, polymorphism, delegation, abstract class, and generic class. Further studies, some of which are currently underway in our laboratory, are needed to further extend, elucidate and elaborate on our proposed list of refactoring patterns. Additional work is also needed to investigate the relationship between refactoring and the analysis or design patterns, particularly those involved in GoF (Gamma et al., 1995). In fact, an existing UML model can be improved by the operation of refactoring, which introduces a design or analysis pattern. This objective could be accomplished by the composition of existing refactoring patterns. (see Fig. 1)

Acknowledgments

The authors would like to express their sincere gratitude to Mr. Anouar Smaoui from the English Language Unit at the Faculty of Science of Sfax, Tunisia for his valuable language editing and polishing services.

References

Abrial, J.R., 1996. The B Book – Assigning Programs to Meanings. Cambridge University Press.

Allem, K., Mens, T., 2007. Refactoring des modèles: concepts et défis. In: Proc. IDM 2007. Hermes Science Publications, Lavoisier.

Baumeister, H., Weber, B., 2013. Agile Processes in Software Engineering and Extreme Programming. In: Proceedings of 14th International Conference, XP 2013. Springer Publishing Company, Vienna, Austria, Incorporated, 2013.

Behm, P., Benoit, P., Meynadier, J.M., 1999. METEOR: a successful application of B in a large project. In: Integrated Formal Methods, IFM99. LNCS, vol. 1708. Springer Verlag, pp. 369–387.

Ben Ammar, B., 2012. Raffinement et Refactoring de spécifications UML: Contribution à l'ingénierie des systmes (PhD thesis). Editions Universitaires Européennes EUE.

Ben Ammar, B., Bhiri, M.T., Benhamadou, A. Pattern de raffinement: Introduction d'une classe intermédiaire ClassHelper. In: Conférence en IngénieriE du Logiciel (Rennes France, 2012).

Ben Ammar, B., Bhiri, M.T., Souquières, J. Schéma de refactoring de diagrammes de classes basé sur la notion de délégation. In: 7ème atelier sur l'Evolution, Réutilisation et Traçabilité des Systèmes d'Information, ERTSI, couplé avec le XXVIème congrès INFORSID (Fontainebleau France, 2008).

Correa, A., Werner, C., 2007. Refactoring object constraint language specifications. Software Syst. Model. 6 (2), 113–138.

Dijkstra, E.W., 1975. Guarded commands, nondeterminacy and formal derivation of programs. Commun. ACM 18 (8), 453–457.

Einarsson, H., Neukirchen, H., 2012. An approach and tool for synchronous refactoring of uml diagrams and models using model-to-model transformations. In: Proceedings of the Fifth Workshop on Refactoring Tools, WRT '12. ACM, New York, NY, USA, pp. 16–23.

Engineering, C.S., 2009. Atelier B 4 – User Manual, Version 4.0.

Fowler, M., 1999. Refactoring: Improving the Design of Existing Code. Addison-Wesley, Boston, MA, USA.

Gamma, E., Helm, R., Johnson, R., Vlissides, J., 1995. Design Patterns. Addison-Wesley, Boston, MA.

Goldsmith, M., 2005. FDR2 User's Manual version 2.82.

Gorp, P., Stenten, H., Mens, T., Demeyer, S., 2003. Towards automating source-consistent UML refactorings. In: Stevens, P., Whittle, J., Booch, G. (Eds.), UML 2003 – The Unified Modeling Language, Lecture Notes in Computer Science, vol. 2863. Springer-Verlag, pp. 144–158.

Hoare, C.A.R., 2004. Communicating sequential processes. Electronic edition edited by Jim Davies.

Idani, A., Ledru, Y., Labiadh, M.-A. Ingénierie dirige par les modèles pour une intgration efficace de uml et b. In: INFORSID 2009 (Toulouse, May 2009).

Ifill, W., Schneider, S.A., Treharne, H., 2007. Augmenting B with control annotations. In: Julliand, J., Kouchnarenko, O. (Eds.), B, Lecture Notes in Computer Science, vol. 4355. Springer, pp. 34–48.

Laleau, R. Conception et développement formels d'applications bases de données (PhD thesis). CEDRIC (CNAM), University of Evry, 2002. Habilitation à diriger des recherches.

Lano, K., Clark, D., Androutsopoulos, K., 2004. Uml to b: Formal verification of object-oriented models. In: IFM, pp. 187–206.

Ledang, H., 2001. Automatic translation from UML specifications to B. In: ASE '01: Proceedings of the 16th IEEE International Conference on Automated Software Engineering. IEEE Computer Society, p. 436.

Ledang, H., Souquières, J., 2002. Integration of uml and B specification techniques: systematic transformation from ocl expressions into b. In: APSEC '02: Proceedings of the Ninth Asia-Pacific Software Engineering Conference. IEEE Computer Society, Washington, DC, USA, p. 495.

Marcano, R., Levy, N., 2002. Using B formal specifications for analysis and verification of UML/OCL models. In: Kuzniarz, L., Reggio, G., Sourrouille, J.L., Huzar, Z. (Eds.), Blekinge Institute of Technology, Research Report 2002:06. UML 2002, Model Engineering, Concepts and Tools. Workshop on Consistency Problems in UML-based Software Development. Workshop

[2] We slightly modified our specification CSP so that it is accepted by ***FDR2***.

Materials. Department of Software Engineering and Computer Science, Blekinge Institute of Technology, pp. 91–105.
Markovic, S., 2008. Model refactoring using transformations (PhD thesis). Lausanne.
Marković, S., Baar, T., 2008. Refactoring ocl annotated uml class diagrams. Softw. Syst. Model. 7 (1), 25–47.
Mens, T., 2006. On the use of graph transformations for model refactoring. In: Joost (Ed.), Generative and Transformational Techniques in Software Engineering, Lecture Notes in Computer Science, vol. 4143. Springer, pp. 215–254.
Mens, T., Gorp, P.V., 2005. A taxonomy of model transformation. In: Proc. International Workshop on Graph and Model Transformation (GraMoT), vol. 152. Elsevier.
Mens, T., Tourwe, T., 2004. A survey of software refactoring. IEEE Trans. Softw. Eng. 30 (2), 126–139.
Mens, T., Taentzer, G., Müller, D., 2007. Challenges in model refactoring. In: Proc. 1st Workshop on Refactoring Tools. University of Berlin.
Mens, T., Taentzer, G., Runge, O., 2007. Analysing refactoring dependencies using graph transformation. Softw. Syst. Model., 269–285.
Meyer, B., 2000. Conception et programmation orientées objet. Eyrolles.
Meyer, E., Souquières, J., 1999. A Systematic Approach to Transform OMT Diagrams to a B Specification. In: FM '99: Proceedings of the World Congress on Formal Methods in the Development of Computing Systems, vol. I. Springer-Verlag, pp. 875–895.
Rasch, H., Wehrheim, H., 2003. Checking Consistency in UML Diagrams: Classes and State Machines. In: Najm, E., Nestmann, U., Stevens, P. (Eds.), Formal Methods for Open Object-based Distributed Systems, LNCS, vol. 2884. Springer, pp. 229–243.
Reimann, J., Seifert, M., Amann, U., 2010. Role-based generic model refactoring. In: Petriu, D., Rouquette, N., Haugen, A. (Eds.), Model Driven Engineering Languages and Systems, Lecture Notes in Computer Science, vol. 6395. Springer, Berlin Heidelberg, pp. 78–92.
Reimann, J., Wilke, C., Demuth, B., Muck, M., Amann, U., 2012. Tool supported ocl refactoring catalogue. In: Proceedings of the 12th Workshop on OCL and Textual Modelling, OCL '12. ACM, New York, NY, USA, pp. 7–12.
Roscoe, A.W., 1994. Model-checking CSP. Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK.
Schwaber, K., Sutherland, J., 2013. The scrum guide.
Shore, J., Warden, S., 2007, The Art of Agile Development, first ed. O'Reilly.
Snook, C., Butler, M., 2004. U2b – a tool for translating uml-b models into b. In: UML-B Specification for Proven Embedded Systems Design.
Sunyé, G., Pollet, D., Le Traon, Y., Jézéquel, J.-M., 2001. Refactoring UML models. In: Proceedings of UML 2001. LNCS, vol. 2185. Springer Verlag, pp. 134–148.
van Kempen, M., Chaudron, M., Kourie, D., Boake, A., Boake, A., 2005. Towards proving preservation of behaviour of refactoring of uml models. In: SAICSIT '05: Proceedings of the 2005 annual research conference of the South African institute of computer scientists and information technologists on IT research in developing countries (Republic of South Africa), South African Institute for Computer Scientists and Information Technologists, pp. 252–259.