Université de Montréal

Un formalisme pour la traçabilité des transformations

par
Mathieu Lemoine

Département d'informatique et de recherche opérationnelle
Faculté des arts et des sciences

Mémoire présenté à la Faculté des arts et des sciences
en vue de l'obtention du grade de Maître ès sciences (M.Sc.)
en informatique

Décembre, 2009

Université de Montréal
Faculté des arts et des sciences

Ce mémoire intitulé:

Un formalisme pour la traçabilité des transformations

présenté par:

Mathieu Lemoine

a été évalué par un jury composé des personnes suivantes:

Michel Boyer,	président-rapporteur
Yann-Gaël Guéhéneuc,	directeur de recherche
Merlo Ettore,	membre du jury

Mémoire accepté le: . . . . . .

RÉSUMÉ

Dans le développement logiciel en industrie, les documents de spécification jouent un rôle important pour la communication entre les analystes et les développeurs. Cependant, avec le temps, les changements de personel et les échéances toujours plus courtes, ces documents sont souvent obsolètes ou incohérents avec l'état effectif du système, *i.e.*, son code source. Pourtant, il est nécessaire que les composants du système logiciel soient conservés à jour et cohérents avec leurs documents de spécifications pour faciliter leur développement et maintenance et, ainsi, pour en réduire les coûts. Maintenir la cohérence entre spécification et code source nécessite de pouvoir représenter les changements sur les uns et les autres et de pouvoir appliquer ces changements de manière cohérente et automatique.

Nous proposons une solution permettant de décrire une représentation d'un logiciel ainsi qu'un formalisme mathématique permettant de décrire et de manipuler l'évolution des composants de ces représentations. Le formalisme est basé sur les triplets de Hoare pour représenter les transformations et sur la théorie des groupes et des homomorphismes de groupes pour manipuler ces transformations et permettrent leur application sur les différentes représentations du système.

Nous illustrons notre formalisme sur deux représentations d'un système logiciel : PADL, une représentation architecturale de haut niveau (semblable à UML), et JCT, un arbre de syntaxe abstrait basé sur Java. Nous définissons également des transformations représentant l'évolution de ces représentations et la transposition permettant de reporter les transformations d'une représentation sur l'autre. Enfin, nous avons développé et décrivons brièvement une implémentation de notre illustration, un plugiciel pour l'IDE Eclipse détectant les transformations effectuées sur le code par les développeurs et un générateur de code pour l'intégration de nouvelles représentations dans l'implémentation.

Mots clés: traçabilité, modèle, code source, théorie des groupes, transpositions.

ABSTRACT

When developing software system in industry, system specifications are heavily used in communication among analysts and developers. However, system evolution, employee turn-over and shorter deadlines lead those documents either not to be up-to-date or not to be consistent with the actual system source code. Yet, having up-to-date documents would greatly help analysts and developers and reduce development and maintenance costs. Therefore, we need to keep those documents up-to-date and consistent.

We propose a novel mathematical formalism to describe and manipulate the evolution of these documents. The mathematical formalism is based on Hoare triple to represent the transformations and group theory and groups homomorphisms to manipulate these transformations and apply them on different representations.

We illustrate our formalism using two representation of a same system: PADL, that is an abstract design specification (similar to UML), and JCT, that is an Abstract Syntax Tree for Java. We also define transformations describing their evolutions, and transformations transposition from one representation to another. Finally, we provide an implementation of our illustration, a plugin for the Eclipse IDE detecting source code transformations made by a developer and a source code generator for integrating new representations in the implementation.

Keywords: traceability, model, source code, group theory, transpositions.

TABLE DES MATIÈRES

LISTE DES FIGURES

LISTE DES SIGLES

API	Interface de programmation (*Application Programming Interface*)
ASM	Machine à état abstrait (*Abstract State Machine*)
AST	Arbre de syntaxe abstraite (*Abstract Syntax Tree*)
DSL	Langage spécifique au domaine (*Domain Specific Language*)
GC	Générateur de code
IDE	Environnement de développement intégré (*Integrated Development Environment*)
JCT	*JavaC Bound AST*
MM	Meta-Modèle
PADL	*Patterns and Abstract-level Description Language*
PT&TT	*Program Transformations and Transformation Transpositions*
Trans	Transformations
UML	Langage de modélisation unifié (*Unified Modeling Language*)
VDM	Méthode de développement de Vienne (*Vienna Development Method*)

REMERCIEMENTS

Je tiens à remercier vivement mon directeur de recherche, Yann-Gaël Guéhéneuc, qui m'a aidé à orienter mes recherches et porté conseil tout au long de ma maîtrise.

Je remercie mes collègues, Julien Tantéri et Stéphane Vaucher, ainsi que tous les autres étudiants du laboratoire pour m'avoir aidé de par leurs idées et conseils durant le déroulement de ma maîtrise et la rédaction de mon mémoire.

Je remercie vivement ma famille et mes amis, de Montréal, de France et d'ailleurs, qui m'ont soutenus et supportés durant ces années d'études.

CHAPITRE 1

INTRODUCTION

1.1 Contexte : traçabilité des changements entre documents et code

Le développement et la maintenance de systèmes logiciels en industrie impliquent souvent plusieurs analystes et développeurs, répartis dans plusieurs équipes, et de plus en plus souvent distribués géographiquement et temporellement. Dans ce contexte, les documents de spécification jouent un rôle important dans la communication entre les différents développeurs. Cependant, durant le cycle de vie du logiciel, ces documents sont modifiés de nombreuses fois par les divers développeurs et les maintenir dans un état cohérent et à jour par rapport à l'état effectif du système, *i.e.*, son code source, est compliqué et coûteux. Ainsi, ils deviennent souvent obsolètes ou incohérents avec le code source ou l'architecture effectifs du système. Pourtant, quand le logiciel atteint la phase de maintenance, ces documents, s'il étaient cohérents et à jour, seraient très utiles et permettraient de réduire les coûts de développement et de maintenance [6]. Afin de maintenir les documents de spécification et le code source à jour, nous avons besoin de décrire chacune de ces deux représentations et de propager ou reporter, pour chaque élément constituant ces représentations, toute modification d'une représentation sur les autres. Nous pourrons ainsi assurer la *traçabilité* entre ces réprésentations [17].

1.2 Contributions

Les contributions de ce mémoire se répartissent en trois catégories : un formalisme mathématique permettant de représenter les transformations des différents composants et de garantir les propriétés désirables et nécessaires au maintien de la cohérence entre les documents de spécification et le code source, l'illustration de ce formalisme grâce à deux types de représentations du système et l'implémentation de cette illustration ainsi que d'outils permettant une utilisation aisée de notre solution. La Figure 1.1 résume les composants de nos contributions ainsi que leurs interactions.

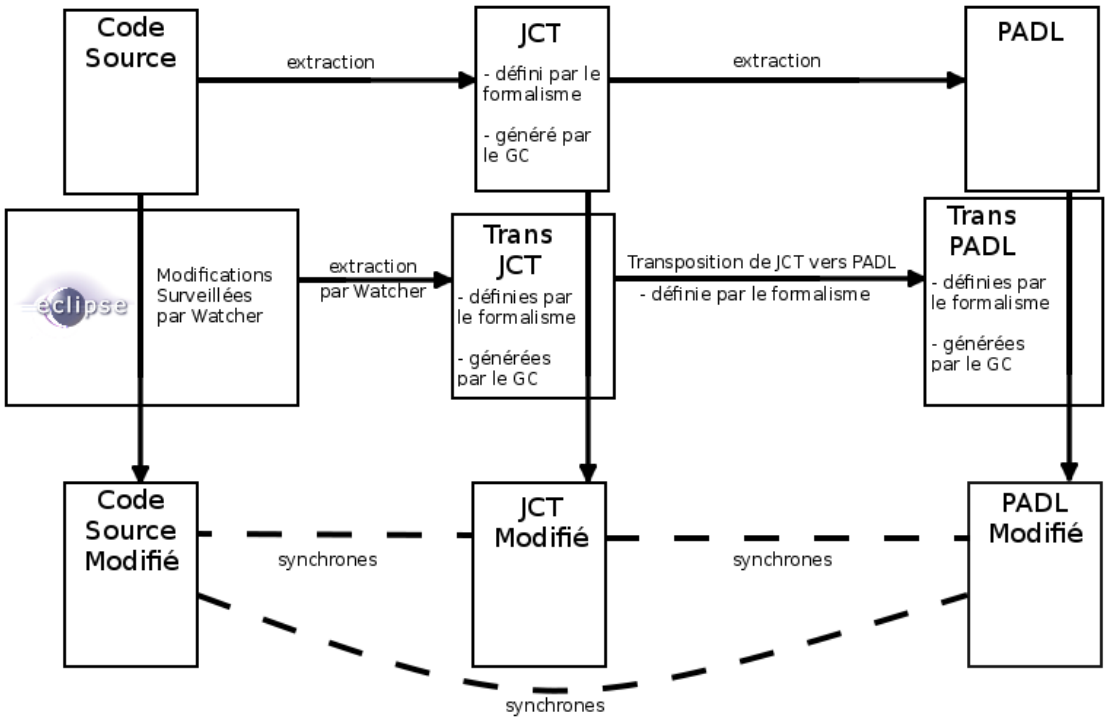

Figure 1.1 – Résumé de nos contribution et de leurs interactions

1.2.1 Formalisme : triplets de Hoare, théorie des groupes et homomorphismes de groupes

Notre approche se base sur un formalisme mathématique permettant de garantir les propriétés qui nous sont nécessaires pour conserver la traçabilité des composants de deux représentations d'un système logiciel. Notre formalisme utilise chaque représentation du système comme un modèle, défini par un méta-modèle. Afin de représenter les modifications et l'évolution de ces modèles, nous proposons, pour chacun d'eux, un ensemble de transformations représentées sous forme de triplets de Hoare [22] qui présentent une structure très proche de celle d'une transformation.

Dans le but de propager les transformations d'un modèle aux autres, notre formalisme s'appuie sur la théorie des groupes et les homorphismes de groupes pour garantir l'intégrité des transformations ainsi que les divers propriétés nécessaires à ces transpositions.

Notre formalisme est centré autour des transformations et des transpositions les considérant comme des entités de première classe. Ce qui nous permet d'exprimer certaines propriétés, telles que l'inversibilité des transformations ou la stabilité des transpositions, qui sont difficiles à représenter via d'autres formalismes.

1.2.2 Illustration : JCT et PADL

En vue de démontrer l'applicabilité de notre formalisme, nous en présentons une application. Cette application est composée de deux méta-modèles permettant de représenter un programme orienté objet en Java selon deux niveaux d'abstraction. Le premier méta-modèle, JCT, est un AST du code source. Le second méta-modèle, PADL [18], est une représentation abstraite du programme, répertoriant les paquetages (*packages*), classes, champs (*fields*) et signatures des méthodes présents dans le programme [3, 18, 19].

Pour chacun de ces deux méta-modèles, nous présentons un ensemble de transformations permettant de représenter les changements sur leurs modèles. Enfin, nous présentons deux transpositions, permettant de propager les transformations sur JCT vers PADL et inversement.

1.2.3 Implémentation : JCT, transformations, transpositions, Watcher et générateur de code

Nous avons développé une implémentation de JCT, des transformations sur JCT et sur PADL, ainsi que des transpositions présentées pour illustrer le formalisme. Dans le but de faciliter la génération de nouvelles transformations, nous avons développé un plugiciel (*plugin*) pour l'IDE Eclipse permettant de générer les modifications textuelles du code source à partir des changements réalisés par les développeurs. Ce plugiciel comporte également un analyseur (*parser*), permettant d'extraire des transformations sur JCT à partir de ces modifications textuelles, proposé en tant que preuve de concept.

Enfin, afin de faciliter l'intégration de nouveaux méta-modèles à notre implémentation, nous avons développé un générateur de code permettant de générer rapidement le

code nécessaire à l'intégration d'un nouveau méta-modèle à notre implémentation.

1.3 Plan

Le second chapitre présente une revue de la littérature liée à nos contributions. Le troisième chapitre de ce mémoire décrit notre formalise ainsi que son illustration tels que présentés dans un article soumis pour le *Journal of Software Maintenance and Evolution: Research and Practice*. Le quatrième chapitre détaille le fonctionnement le plugiciel développé pour Eclipse ainsi que l'exploitation des données produites par celui-ci. Le cinquième chapitre fournit un aperçu du générateur de code développé pour faciliter l'intégration de nouveaux méta-modèle dans notre implémentation. Enfin, le sixième et dernier chapitre conclut ce mémoire et présente des pistes de recherches futures.

CHAPITRE 2

ÉTAT DE L'ART

Diverses approches ont été présentées pour étudier l'évolution des systèmes logiciels. Les versions successives d'un même système logiciel sont souvent conservées dans un système de contrôle de version. Diverses analyses portant sur les données historiques contenues dans ces dépôts ont été présentées [20, 35, 36]. Parmi ces approches, on peu souligner l'analyse d'impact des changements [21, 23, 34] qui permet d'étudier l'impact global de modifications locales dans un système logiciel. Les analyses retraçant les effets des modifications locales d'un système logiciel à travers ses divers composants sont des analyses de traçabilité.

La majorité des travaux portant sur la traçabilité de composants entre représentations d'un même sytème logiciel n'ont pour objectif principal que de prévenir le développeur de la destruction des liens de traçabilité [31], ou de retrouver ces liens après leur destruction [25]. Par exemple, le travail de Reiss [31] fournit un environnement permettant de définir et de vérifier plusieurs contraintes de traçabilité entre différents documents, (*e.g.*, une interface dans le code source doit apparaître dans le diagramme de classe UML) et informe le développeur si la traçabilité est menacée par une contrainte qui n'est pas respectée. Notre approche vise à préserver automatiquement la traçabilité en conservant les documents à jour et cohérents.

Une grand partie des travaux sur la traçabilité concerne des documents non structurés [4] ou annexes (*i.e.*, qui ne sont pas des représentations directes du système logiciel, tels que les tests unitaires) [32]. Par exemple, le travail de Rompaey et Demeyer [32] présente un ensembles de stratégies basées sur l'analyse statique permettant d'extraire des liens de traçabilité entre le code source et les tests unitaires. Notre approche se concentre sur les différentes représentations structurées d'un système.

Les transformations sur le code source ont d'abord été étudiées en tant que réusinage (*refactoring*) [12] pour la première fois par Opdyke [30] et Fowler [15]. Cependant, les réusinages ont une forte granularité (*coarse-grained*) et ne représentent que des mo-

difications spécifiques, ne modifiant pas le comportement d'un programme, alors que les transformations ont une granularité plus fine et permettent de représenter n'importe quelle modification. De plus, leurs travaux ont principalement présenté un catalogue de ces réusinages et ils n'ont publié aucune formalisation s'appuyant et satisfaisant des propriétés mathématiques.

Les transformations sont souvent représentées en tant que transformations de graphes [7, 27]. Cette représentation, bien qu'intuitive, ne permet pas d'exprimer aisément nos propriétés, contrairement à notre représentation basée sur l'algèbre abstraite. De plus les deux représentations ne sont pas conflictuelles, les triplets de Hoare pouvant être considérés comme une abstraction des transformations de graphes.

Abrial [2] a indiqué que les méthodes formelles ne sont pas toujours simples à appliquer en industrie. Cependant, contrairement à ASM [8], B [1] ou VDM [24], notre but n'est pas de décrire une méthode de travail mais d'utiliser un formalisme mathématique pour garantir les propriétés utilisées par des outils. Ainsi, l'utilisateur final n'a pas besoin de connaître les détails du formalisme pour pouvoir utiliser les outils créés à partir de notre formalisme. Par exemple, aucune connaissance du formalisme n'est nécessaire pour utiliser notre plugiciel Eclipse Watcher.

Oda et Saeki [29] présentent une approche centrée sur le contrôle de versions. Ils représentent un modèle comme un graphe et définissent un ensemble d'opérations telles que la création, la suppression et la mise-à-jour des constituants d'un modèle afin de conserver un historique des différentes versions d'un modèle. Cependant, ces transformations ne peuvent être reportées sur d'autres représentations du système. Une évolution de notre formalisme pourrait être utilisée comme base d'un gestionnaire de versions prenant en compte la structure du source code, utilisant des propriétés similaires à la théorie des rustines (*patch*) de DARCS [10, 11], telles que la commutativité.

Moha [28] représente les transformations applicables sur plusieurs méta-modèles simultanément. Cependant, cette stratégie nécessite que les propriétés des transformations soient garanties pour chacun des méta-modèles. De plus, cette restriction revient à lier ensemble les différents méta-modèles et nous oblige à modifier les transformations elle-même pour intégrer un nouveau méta-modèle à notre approche. Nous préférons donc dé-

finir des transpositions explicites d'un méta-modèle à l'autre. Cette stratégie nous permet de définir des transformations plus simples. De plus, l'intégration d'un nouveau méta-modèle ne requiert que la définition de transpositions entre le nouveau méta-modèle et ceux avec lesquels il doit interagir.

Le travail le plus proche de celui présenté dans ce mémoire a été effectué par Diskin [13]. Nos deux approches comportent certains objets et propriétés très similaires. Ainsi, les systèmes de synchronisation diagonale (*diagonal synchronization systems*) de Diskin sont très similaires dans leur utilisation à une paire de transpositions, ces deux objets permettent d'effectuer les mises à jour entre deux modèles afin d'en assurer la cohérence. Ils présentent également des propriétés similaires, ainsi la propriété de reversibilité *(Undo)* est requise pour nos transformations. De même, la propriété d'Hippocrate (*Hippocraticness*) est assimilable à notre définition de stabilité. Cependant, l'approche mathématique de Diskin est centrée sur les modèles en tant que entités de première classe et représente les transformations en tant que fonctions travaillant sur des modèles. Notre approche représente non seulement les modèles mais également les transformations comme des entités de première classe. De plus, le formalisme de Diskin est basé sur l'analyse de fonctions, qui fait partie de la théorie des ensembles ; notre formalisme, quand à lui, est basé sur la théorie des groupes, qui relève du domaine de l'algèbre abstraite.

En résumé, notre approche vise à préserver automatiquement la traçabilité en conservant les divers documents composant un système logiciel à jour et cohérents. Notre formalisme, se concentrant sur des représentation structurées et directes du système, permet une caractérisation mathématique des transformations en utilisant l'algèbre abstraite. Chaque méta-modèle est considéré séparément des autres, ce qui permet d'ajouter ou d'écarter facilement un méta-modèle sans avoir à modifier les autres ou les transformations.

CHAPITRE 3

FORMALISME

Nous introduisons et illustrons maintenant notre formalisme. Ce chapitre est un article soumis au *Journal of Software Maintenance and Evolution: Research and Practice*. Il décrit ainsi notre formalisme, l'illustre sur un exemple, décrit son implémentation et reprend l'état de l'art (présenté dans le Chapitre 2).

A Formalism for Transformations Traceability

Mathieu Lemoine, Yann-Gaël Guéhéneuc
Ptidej Team, DIRO, Université de Montréal,
Montréal, QC, Canada

March 16, 2010

Abstract

When developing of software system in industry, system specifications are heavily used in communication among analysts and developers. However system evolution, employee turn-over, and shorter deadlines, lead these documents to either not be up-to-date or not be consistent with the actual system source code. Yet, having up-to-date documents would greatly help analyst and developers and reduce development and maintenance cost. Therefore, we need to keep these documents up-to-date and consistent.

We propose a novel mathematical formalism to describe and manipulate the evolution of these documents. The mathematical formalism is based on Hoare triple to represent the transformations, and group theory and groups homomorphisms to manipulate these transformations and apply them on different representations.

We illustrate our formalism using two representation of a same system: PADL, an abstract design specification (similar to UML), and JCT, an Abstract Syntax Tree for Java. We define also transformations describing their evolutions and transformations transposition from one representation to another. Finally, we provide an implementation of our illustration and present a working example of our approach.

1 Introduction

Development and maintenance of software systems in industry often involve many analysts and developers in distributed teams. Specification documents allow communication among the different developers involved in the life-cycle of the system. However, these documents are usually either not up-to-date or not consistent with the actual source code and design of the system. Yet, when the software enters its maintenance phase [6], these documents if they were consistent and up-to-date could be very useful.

Therefore, we need an approach to keep automatically the artifacts composing theses documents up-to-date. To fulfill this need, we must achieve traceability. Traceability is the ability to describe the links among artifacts and follow

automatically their life to keep different documents up-to-date and to retrieve any previous version of these documents [14].

In this paper, we contribute to research on traceability with an approach, composed of a formalism and its implementation. Our formalism provides abstract representations, meta-models, of a system and represents their modifications with transformations. Different meta-models describe different abstraction levels. To keep all these corresponding models up-to-date and consistant despite the different level of abstraction, we define mathematical properties on these transformations and operations to propagate them from one model to the other.

1.1 Running Example

We illustrate our approach on a light HTTP server, whose development team is composed of an analyst (A) and a developer (D). While the analyst, upon receiving change requests sent by a client, applies global, abstract modifications on the design model, the developer is filling-in the code and implementing features.

Our example assumes the existence of a first draft of the design of the system made by the analyst, as shown in Fig.4; we provide an excerpt of source code before the modifications in Fig.1. When A and D are working on the system, discrepancies may occur between the design and the source code models; our approach ensures the consistency of the documents by keeping them up-to-date. After the modifications have been applied on the system, both the design and the excerpt of source code evolved, as shown in Fig. 5, 2 and 3.

Let us assume a list of change requests asked by the client:

– An easier configuration of the server;
– The support for dynamically generated content.

To answer these requests, the analyst and the developer modify the system as follow:

1. To support an easier configuration of the server:
   (a) Replace the constructor of the *Listener* by a method factory;
   (b) Create a *Item* class, later renamed to *Page*, to represent each request received by the server so that the handling of each request can be customized;
2. To support dynamically generated content:
   (a) Replace the *Answerer* class by a hierarchy of interfaces and classes;
   (b) Create an abstract factory so that each request can be attribued to different response strategy (static or dynamic content).

The comprehensive list of modifications on the server by A and D are available in a longer technical report [19]. For the sake of simplicity, we illustrate our approach by studying how modifications are transposed between the source code and the design model. Three scenarios are possible:

1. A transformation that is equivalent or almost equivalent in both models, *i.e.*, whose transposition is trivial;
2. A transformation that can be expressed only in one model, because of the difference of abstraction level between models;
3. A transformation that can be represented in both models, but whose transposition is non-trivial.

We illustrate these three scenarios by presenting three transformations:

1. Renaming a class, during modification 1.b. The analyst renames the class `Item` into `Page`;
2. Removing a statement, during modification 1.a. The developer removes the first statement of the body of the `getListener` method within the `Listener` class;
3. Adding a compilation unit in a package, during modification 2.a. The developer adds the compilation unit `IAnswerer.java`, containing the interface `IAnswerer`, in the package `httpserver`

We illustrate our approach by detailling how our implementation keeps the different models up-to-date when these three modifications are applied in the following.

1.2 Outline

This paper is organised as follows. Section 2 describes the different objects that we manipulate to maintain the traceability between design document and source code and their properties. Section 3 presents the exact mathematical description of our formalism. Section 4 introduces our implementation. Section 5 illustrate our formalism and our approach by presenting our implementation and studying our running example. Section 6 presents how our approach contribute with respect to related work. Finally, Section 7 concludes and suggests future work.

```
1     public class Answerer {
2       private final DataOutputStream socket;
3
4       public Answerer(final DataOutputStream socket) {
5         this.socket = socket;
6       }
7
8       public void generate() {
9         try {
10          this.socket.write("200␣OK\n\nIt␣works␣!");
11        } catch(final IOException e) {
12          e.printStackTrace();
13        }
14      }
15    }
```

Figure 1: excerpt of original source code [file Answerer.java]

```
1     public interface IAnswerer {
2       public void generate();
3     }
```

Figure 2: excerpt of source code after modifications [file IAnswerer.java]

```
1     public class AnswererFactory {
2       public IAnswerer createAnswerer(final Page request) {
3         final File requestedFile(request.getRequestedURI());
4
5         if(requestedFile.isFile()) {
6            if(requestedFile.getName().endsWith(".html")) {
7             return new StaticAnswerer(requestedFile);
8            } else {
9             return new DynamicPageAnswerer(requestedFile,
10                                           true);
11        } else {
12         return new DynamicPageAnswerer(requestedFile,
13                                        false);
14        }
15      }
16    }
```

Figure 3: excerpt of source code after modifications [file AnswererFactory.java]

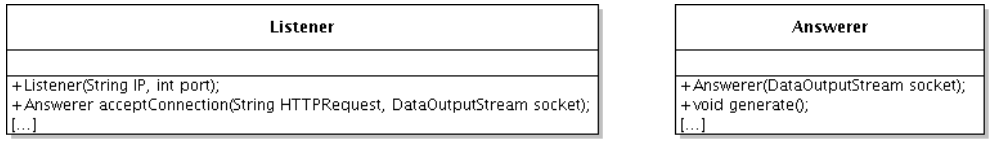

Figure 4: First-draft, original design

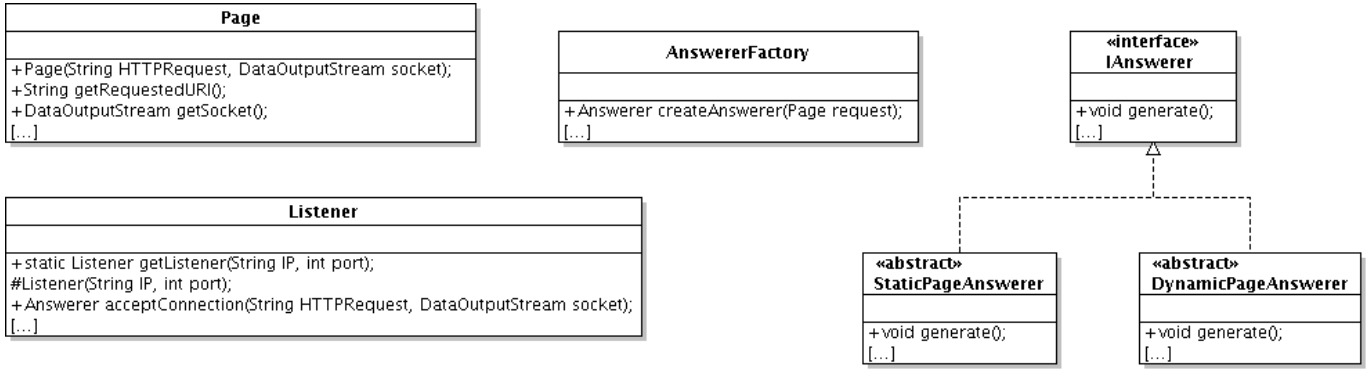

Figure 5: Design model after the changes

2 Background

In this section, we present our representations of software system representations, *i.e.* models and meta-models, transformations and transformations transpositions and their properties.

2.1 Meta-models

A meta-model is the specification of a type of models, *e.g.*, UML class diagrams are described by the MOF meta-model, it is a model for models [9]. A meta-model is designed to describe some artifacts of the system at some level of abstraction. A meta-model is composed of a structural definition, the structural meta-model, and a set of constraints (*e.g.*, the uniqueness of identifiers) called referential integrity.

Depending on what developpers and analysts need in the models, different meta-models can be designed, *e.g.*, UML, language specification, entity-relationship diagrams.

In the following, we describe meta-models by using these following conventions:

- the "`is-a`" specification allows to factor out the specifications of similar constituents. A declaration such that "A `is-a` B" means that A inherits B.
- the attributes preceded by the term "`has-a`" denote a composition relationship between the constituent and the value(s) of the attribute. The constituent is the container of the value(s). These value(s) must be constituent themselves, and a value may be contained at most once.
- the attributes specified in **bold** are identifiers for the constituents, they must be unique within a model;
- the type of each attribute is specified within curly bracket ("{" and "}");
- the default values are specified between square brackets ("[" and "]");

2.2 Models

A model is a reprensentation of a system and it must comply with a meta-model. When a transformation is applied on a model, it produces a new version of the model. During the application of the transformation, it is possible temporarily to break some constraints of the referential integrity. In any other context, a model should be viewed as stable:

Definition 2.2.1 (*Stable model*)**.** A stable model is a model that is not being modified by a transformation. Such a model must comply to its meta-model (*i.e.*, including referential integrity).

2.3 Transformations on Models

We call transformation the reification of a model modification. It is important to note that unlike semantic preserving transformations [5], transforma-

tions do not preserve the semantic of the program. Every transformation must be specified with respect to a specific meta-model. We say that the transformations *assumes* the meta-model. The specification of a modification is a template transformation containing free variables (its parameters). We create a transformation representing a concrete modification of system software representation by instantiating the template, *i.e.*, by binding its parameters to real values. For example, given a transformation template *SetClassName* assuming some meta-model, its parameters would be a way to identify the class being renamed and the new name of the class.

Wherever no distinction is required between a transformation and the template used to create it, or when the distinction is trivial within a context, we use the generic term "transformation".

Each transformation template is composed of the following constituents:

- assumed meta-model: a transformation is defined *under* a meta-model and applied *on* a model;
- parameters: data characterising an instance of the transformation;
- invariants: conditions on the model that must be always true including during a transformation; since the transformation may temporarily break referential integrity, it is sometime useful to require a few unbreakable invariants for a transformation;
- pre-conditions: guards on the model that must be true before applying the transformation;
- post-conditions: conditions on the model that must be true after applying the transformation;
- actions: concrete modifications made to the model constituents to concretly apply the transformation (*e.g.*, "Change attribute `name` in `class` *package.oldName*, from *oldName* to *newName*").

Transformations on a model can require many actions (*e.g.*, rename a parameter, create a new method and move the parameter in it could be one transformation), and it is impossible to specify all of them before hand. Thus, we define just a few primitive transformations and combine them in (finite) sequences.

Definition 2.3.1 (*Primitive transformation*)**.** A primitive transformation is a transformation with few actions, used to represent a simple modification to a model. Definitions of "few" and "simple" are left unspecified on purpose to accommodate different definitions in different contexts.

Definition 2.3.2 (*Sequencing*)**.** Sequencing is the ability to combine (a finite number of) transformations into complex transformations, so that complex transformations are easier to consider.

To retrieve a previous version of a model, transformations must be reversible and have associative sequencing. Moreover, implementations should have a relationship denoting equivalence of two (sequences of) transformations.

Property 2.3.1 (*Reversibility*)**.** Reversibility of transformations means that for each transformation, another one reverts its modifications.

Property 2.3.2 (*Associativity*)**.** Associativity means that given three transformations a, b and c, and the sequencing operation $\rightarrow$: "$(a \rightarrow b) \rightarrow c$" must give the same result as "$a \rightarrow (b \rightarrow c)$".

2.4 Traceability: Transpositions of Transformations

Our goal is to keep different models of the system up-to-date and constitent. We therefore must transpose our transformations from one meta-model to another one, so we can apply transformations on differents models.

A transposition must preserve the sequencing of transformations so we are able to manipulate and combine transposed and other transformations seamlessly, we call this property coherence.

Property 2.4.1 (*Coherence*)**.** Coherence of transpositions with respect to the sequencing of transformation is expressed as:

$$\{\{a/MM1\} \rightarrow \{b/MM1\}\} \Rightarrow MM2$$
$$\text{is equivalent to } \{\{a/MM1\} \Rightarrow MM2\} \rightarrow \{\{b/MM1\} \Rightarrow MM2\}$$

where $\{a/MM1\}$ means a transformation a was defined under a meta-model $MM1$, $\rightarrow$ represents the sequencing and $\Rightarrow$ the transposition to another meta-model. We can paraphrase this formula as "transposing the transformation a sequenced with b under MM2" is equivalent to "a transposition of a under MM2 sequenced with a transposition of b under MM2".

Transpositions must maintain transformation semantic as much as possible, *e.g.*, a transformation renaming a class in the source code model should be transposed to a transformation renaming a class in the design model, and vice-versa. To ensure this kind of relationship among transformations on each side of a transposition, the more usual mathematical property is bijectivity:

Property 2.4.2 (*Bijectivity*)**.** Bijectivity of a transposition from a source meta-model to a destination meta-model means that it is always possible to compute deterministically the transformation under the source meta-model from a transposed transformation.

However, bijectivity in some cases is unrealistic, for example, most transformations on the body of a method do not affect the design model. Such transformations should be transposed to an equivalent transformation under the design meta-model that have no effect. However, given a pair of transposition, once a transformation has been transposed, it can be described by both models and should not be modified by subsequent back and forth transpositions. We name this propery stability.

Property 2.4.3 (*Stability*)**.** Stability is a weaker form of bijectivity. Stabiliy means bijectivity after the first application of a pair of transposition, but does not put any other restriction on the result of transposition in general.

However, some transformations can still not be expressed due to the different abstraction levels of different models. To address these transformations, we define a special transformation, identity, under each meta-model:

Definition 2.4.1 (*Identity*)**.** The identity transformation of a meta-model is a special transformation under a meta-model, it is unique and have no effect.

3 Formalism for Transformations

In this section, we bind the properties in Section 2 with mathematical properties. We also present how we use Hoare triples, group theory, and group homomorphisms to ensure the required properties. Stability is a constraint that is not usual in mathematics and we therefore formally define it in the following.

We denote the set of all the transformations defined under a meta-model MM, as $\mathbb{T}_{MM}$.

3.1 Transformations as Hoare Triples

We represent each transformation as a Hoare triple [17]. A Hoare triple is composed of a pre-condition, an action, and a post-condition. It is a representation of actions having some guards (pre-condition), and effects (post-condition). A Hoare triple is noted as :

"{pre-condition}action{post-condition}"

Pre-/post-conditions are required so that we can use a same transformation on many models, as long as they follow the same meta-model and required guards. The pre-/post-conditions are a conjunction of predicates. Conjunctions of predicates being themselves predicates, the parts of the conjunction are called sub-predicates.

The invariants, although present in transformations specifications, are not part of the Hoare triple formalism. We include them in both pre-conditions and post-conditions so that they be always required.

For example, Renaming a class, represented by the transformation *SetClassName*, is modelised as:

$\{E[C_0], \overline{E}[C_1]\}$Change attribute `name` in `class` C_0 from N_0 to $N_1\{\overline{E}[C_0], E[C_1]\}$

with C_0 the class renamed with its old name N_0 and C_1 the same class with its new name N_1. $E[C]$ (resp. $\overline{E}[C]$) is the predicate denoting the existence (resp. non-existence) of C.

An instantiation of the transformation may be, for example,
"$SetClassName[C_0/httpserver.Item, C_1/httpserver.Page, N_0/Item, N_1/Page]$":

$\{E[httpserver.Page], \overline{E}[httpserver.Item]\}$ Change attribute `name` in `class` *httpserver.Page* from *Page* to *Item* $\{\overline{E}[httpserver.Page], E[httpserver.Item]\}$

The only way to represent the effect of a transformation is to list the modifications to the model. A strict identity relationship would be useless. Therefore, an equality relationship is used and is defined as the following equivalence relationship:

$$\text{Let us assume } a \text{ as } \{Pr_a\}A\{Po_a\}, b \text{ as } \{Pr_b\}B\{Po_b\} \in \mathbb{T}_{MM}, \qquad (3.1.1)$$
$$a = b \Longleftrightarrow \{Pr_a \Leftrightarrow Pr_b \wedge Po_a \Leftrightarrow Po_b\}$$

where $\Leftrightarrow$ represents the equivalence of predicates: $p_a \Leftrightarrow p_b$ means that p_a implies p_b and p_b implies p_a.

Logic negation of predicates is defined for any predicate and is denoted as not or $\neg$.

Our properties infer that our equivalence relationship is a mathematical relationship of equivalence:

$$\forall a \in \mathbb{T}_{MM},\ a = a \qquad (3.1.2)$$
$$\forall a, b \in \mathbb{T}_{MM},\ a = b \Longleftrightarrow b = a \qquad (3.1.3)$$
$$\forall a, b, c \in \mathbb{T}_{MM},\ a = b \wedge b = c \Longrightarrow a = c \qquad (3.1.4)$$

3.2 Set of Transformations as Groups

Our transformations need reversibility, associativity, and coherence as presented above. We used groups [12] and group homomorphisms to represent them because they provide almost every property we need and no unneeded property.

We considered other formalisms: on the one hand, monoid formalism would not provide reversibility Prop. 2.3.1 and therefore neither this formalism nor any weaker formalism fits our need; on the other hand, abelian group formalism would include total commutativity of the sequencing that is neither a required property nor one that could be useful a priori, therefore neither this formalism nor any stronger formalism fit our need. Consequently, groups [12] and group homomorphisms are the best suited theory.

For any meta-model MM, the associated group of transformations is composed by the set of transformations defined under it, $\mathbb{T}_{MM}$, and an internal operation, sequencing, represented by $\circ$. In group theory, primitives transformations (Def. 2.3.1) are called group generator.

Moreover, in every group, an (unique) object, the identity element, $\mathbb{I}$, is the neutral element for the intern operation. The unicity of $\mathbb{I}$ is inferred from Eq. 3.2.2 and Eq. 3.2.5.

In every groups, given a meta-model MM, the following properties are al-

ways respected:

$$\forall a, b \in \mathbb{T}_{MM},\ a \circ b \in \mathbb{T}_{MM} \tag{3.2.1}$$

$$\forall a \in \mathbb{T}_{MM},\ a \circ \mathbb{I} = \mathbb{I} \circ a = a \tag{3.2.2}$$

$$\forall a \in \mathbb{T}_{MM},\ \exists!\ a^{-1} \in \mathbb{T}_{MM} \text{ st } a \circ a^{-1} = a^{-1} \circ a = \mathbb{I} \tag{3.2.3}$$

$$\forall a, b \in \mathbb{T}_{MM},\ (a \circ b)^{-1} = b^{-1} \circ a^{-1} \tag{3.2.4}$$

$$\forall a, b, c \in \mathbb{T}_{MM},\ (a \circ b) \circ c = a \circ (b \circ c) \triangleq a \circ b \circ c \tag{3.2.5}$$

Each equation paraphrases a definition or guarantees a property defined in Section 2:

- Primitive Transformations and Sequencing (Def. 2.3.1 and Def. 2.3.2):
  The ability to specify any transformation by specifying only a few class of group generators (primitive transformations) and combine them in sequences is provided by Eq. 3.2.1;
- Associativity (Prop. 2.3.2):
  Associativity is described by Eq. 3.2.5;
- Identity (Def. 2.4.1):
  $\mathbb{I}$ is the `identity` transformation, it does not modify any transformation with which it is sequenced or any model on which it is applied. Its existence and unicity are inferred by Eq. 3.2.2 and Eq. 3.2.5.
- Reversibility (Def. 2.3.1):
  The ability to reverse any transformation (primitive or not) is provided by Eq. 3.2.3 and Eq. 3.2.4. The unicity of the inverse is guaranteed by Eq. 3.2.2 and Eq. 3.2.5.

3.3 Transpositions as Group Homomorphisms

We defined transformations transpositions as group homomorphisms to propagate transformations from one model to another. A group homomorphism is a function from one group to another one that preserves the whole structure of the group.

Given two meta-models MM_1 and MM_2, one homomorphism $F : \mathbb{T}_{MM_1} \rightarrow \mathbb{T}_{MM_2}$, and two transformations r_1 and r_2 under MM_1:

$$F(r_1 \circ r_2) = F(r_1) \circ F(r_2) \tag{3.3.1}$$

which is exactly the mathematical formulation of the property of coherence (Prop. 2.4.1).

Equivalence part of the structure of the group. Hence it must be preserved by group homomorphisms, given the same objects as in Eq. 3.3.1:

$$r_1 = r_2 \Longleftrightarrow F(r_1) = F(r_2) \tag{3.3.2}$$

Composition of group homomorphisms is represented by $\bullet$.

We can formally describe stability (Prop. 2.4.3), given two meta-models MM_1 and MM_2 and two homomorphisms $F : \mathbb{T}_{MM_1} \rightarrow \mathbb{T}_{MM_2}$ and $G : \mathbb{T}_{MM_2} \rightarrow$

$\mathbb{T}_{MM_1}$ as:

$$\forall t \in Im(F) = \{F(x), x \in \mathbb{T}_{MM_1}\} \subseteq \mathbb{T}_{MM_2},\ F \bullet G(t) = t$$
$$\forall t \in Im(G) = \{G(y), y \in \mathbb{T}_{MM_2}\} \subseteq \mathbb{T}_{MM_1},\ G \bullet F(t) = t$$

This definition implies that F and G define a bijective relationship between $Im(F)$ and $Im(G)$. However the relationship does not stand between $\mathbb{T}_{MM_1}$ and $\mathbb{T}_{MM_2}$ in general:

$$\mathbb{T}_{MM_1} \neq Im(G) \Longrightarrow \exists t \in \mathbb{T}_{MM_1} \backslash Im(G), t' \in Im(G) \text{ s.t. } t' = G(F(t))$$

for such a pair (t, t'), $t \neq t'$ but $F(t) = F(t')$. F is therefore not injective on $\mathbb{T}_{MM_1}$, hence not bijective. The same reasoning can be applied on G.

We denote the set of all homomorphisms from $\mathbb{T}_{MM1}$ to $\mathbb{T}_{MM2}$ by $\mathcal{T}_{MM1 \rightarrow MM2}$.

4 Proof of Concept

In this article, we give only an overview of `PADL`, `JCT`, their transformations and transpositions. A complete description is available in a longer technical report [19].

4.1 `PADL`: Meta-model of the Design Models

The `PADL` meta-model has been developed to represents motifs and abstract designs [3,15,16]; it is the meta-model which we use to describe the model of the design of a system. `PADL` is a meta-model which describes abstract constituents of an object-oriented program (*i.e.*, packages, classes, method signatures and fields) but no implementation detail, and an API to manipulate them.

We now define a subset of the `PADL` meta-model and a set of predicates and transformations. We voluntarily omit binary class relationship that we will consider in a future work. We describe each constituent of `PADL` using the format defined in Section 2.2. For example, we specify `Constituent`, the root node of the `PADL` class hierarchy as follow:

`abstract` Constituent
- **Name** {String}
- is Final? {boolean} *[false]*
- is Abstract? {boolean} *[false]*
- is Private? {boolean} *[false]*
- is Public? {boolean} *[false]*
- is Protected? {boolean} *[false]*
- is Static? {boolean} *[false]*
- `has-a` Constituents {Set of Constituents} *[{empty}]*

To ensure referential integrity, we assume that two classes are always part of any `PADL` model: `Object`, the root of the inheritance tree for all classes, and `Throwable`, the root of the inheritance tree for exceptions.

4.1.1 PADL Predicates

We present the formal predicates and their properties using an inductive predicate logic syntax.

Definition

- *Existence of a constituent*: $E[C]$ This predicate evaluates to **true** if the constituent C can be found in a given model M. C is a the constituent, referenced by its path, *e.g.*, for a class, it is its fully qualified name.
- *Value of an attribute of a constituent*: $V[C, A, V]$ This predicate evaluates to **true** if the constituent C can be found in a given model M and if the value of its attribute A is V.
- *Collection membership of an attribute of a constituent*: $in[C, A, V]$ This predicate can be applied only for attribute A that have a type "set". It evaluates to **true** if the constituent C can be found within a given model M and if its attribute A contains the asked value V.

Properties

Predicates can be implied from one or more other predicates, to prove implication or equivalence of predicates (*e.g.*, in the case of equality):

$$V[C, a, v] \Rightarrow E[C], \quad \forall C, a, v$$
$$in[C, a, v] \Rightarrow E[C], \quad \forall C, a, v$$

4.1.2 PADL Primitive Transformations

Eq. 3.2.1 asserts that we can specify the set $\mathbb{T}_{\texttt{PADL}}$ with just a few templates of primitive transformations, any other transformation being created by sequencing the existing transformations. For example, we describe SetClassName as:

SetClassName
- Parameters: an existing class path, a path to the class after its renaming, its (old) name and a new name.
- Invariants: **NONE** {}
- Pre-condition: the class, with its old name, exists, and the class, with its new name, does not exist. $\{E[ClassWithOldName], \neg E[ClassWithNewName]\}$
- Post-condition: the class, with its new name, exists, and the class, with its old name, does not exist. $\{\neg E[ClassWithOldName], E[ClassWithNewName]\}$
- Action: replace the old name by the new one.

4.2 JCT: Meta-model of the Source Code

JCT is based on the javac compiler internal AST, annotated with type and reference information. It provides a structured and easy-to-use representation of the source code. JCT provides all implementation details, including statements (*e.g.*, method body, field initialisation) and compilation units (basically equivalent to a source code file), which does not exist in PADL.

4.2.1 JCT Meta-model

To ensure referential integrity, we assume that four classes are always part of the model: Object, which is the root of the inheritance tree for all classes; Class, which represents the class meta-object; Void, which represents the void object type; and, Throwable, which is the root of the inheritance tree for exceptions.

We describe each constituent of JCT using the format defined in Section 2.2. For example, we specify Import, the constituent representing import statements in Java source code file, as follow:

Import
- **Imported Constituent** {Importable}
- **isStatic** {boolean}
- **isOnDemand** {boolean}

An Importable constituent is a Package, a Class or Interface, or a Static Class Member (*i.e.*, a Static Member Class, a Static Field or a Static Method). These constituents are part of JCT and formaly described in a longer technical report [19].

4.2.2 JCT Predicates

We present the formal predicates and their properties we use to define our transformations under JCT.

Definition

The predicates are same as for PADL, plus:
- *Collection membership of an attribute of a constituent*: $in[C, A, V]$ This predicate is extended so it can be applied for attribute A that have a type "list" too. The semantic is preserved.
- *List membership of an attribute of a constituent*: $at[C, A, V, I]$ This predicate can be applied only for attribute A that have a type "list". It evaluates to **true** if the constituent C can be found within a given model M and if its attribute A contains the asked value V at the asked index I.

The predicate $at[C, A, V, I]$ does not exist in PADL because the index of a value in a "set" attribute is never mandatory to locate a constituent in PADL. In JCT, some constituents can only be located by their index (*e.g.*, the n^{th} statement of a method).

Properties

Predicates can be implied from one or more other predicates, to prove implication or equivalence of predicates (*e.g.*, in the case of equality), the rules are the same as in PADL, plus:

$$\begin{aligned}
at[C,a,v,i] &\Rightarrow & E[C], & & \forall C,a,v,i \\
at[C,A,V,i] &\Rightarrow & in[C,A,V], & & \forall C,A,V,i \\
at[C,A,W,I] \wedge V \neq W &\Rightarrow & \neg at[C,A,V,I], & & \forall C,A,V,W,I \\
at[C,A,V,J] \wedge J \neq I &\Rightarrow & \neg at[C,A,V,I], & & \forall C,A,V,I,J
\end{aligned}$$

4.2.3 JCT Primitives Transformations

As in Section 4.1.3, we defined a set of primitive transformations for JCT. For example, we describe SetClassName as:

SetClassName
- Parameters: an existing class path, a path to the class after its renaming, its (old) name and a new name.
- Invariants: **NONE** {}
- Pre-condition: the class, with its old name, exists, and the class, with its new name, does not exist. $\{E[ClassWithOldName], \neg E[ClassWithNewName]\}$
- Post-condition: the class, with its new name, exists, and the class, with its old name, does not exist. $\{\neg E[ClassWithOldName], E[ClassWithNewName]\}$
- Action: replace the old name by the new one.

4.3 PADL/JCT: an Example of Group Homomorphism

The last part of our implementation is the specification of the transpositions from JCT to PADL and vice-versa.

4.3.1 Transposition from JCT to PADL

Transformations not involving compilation units are transposed to their JCT counterpart.

Transformations from JCT to PADL models not modifying root node, packages, compilation unit, classes, methods or fields are transposed to $\mathbb{I}_{\texttt{PADL}}$, because statements are not represented in PADL.

Compilation units are not represented in PADL, but packages are. Therefore, we add the set of classes in the involved compilation unit to these transformations. Transformations `AddPackageCompilationUnit` are transposed to a sequence of `AddPackageClasss` and, likewise, transformations `RemovePackageCompilationUnit` are transposed to a sequence of `RemovePackageClasss`.

Transformations `AddCompilationUnitClass` are transposed to `AddPackageClass` and, likewise, transformations `RemoveCompilationUnitClass` are transposed to `RemovePackageClass`.

4.3.2 Transposition from `PADL` to `JCT`

Transformations not involving the relationship between packages and packages are transposed to their `PADL` counterpart.

We add the name of the file of the compilation unit to transformations `AddPackageClass` and `RemovePackageClass` and they are respectively transposed to sequences {`CreateCompilationUnit` ∘ `AddPackageCompilationUnit` ∘ `AddCompilationUnitClass`} and {`RemoveCompilationUnitClass` ∘ `RemovePackageCompilationUnit` ∘ `DeleteCompilationUnit`} transformations.

4.4 Running Example

Using our formalism as presented in Section 3 and its instantiation and implementation presented in Sections 4.1 to 4.3, we extract two lists of transformations. The comprehensive lists of transposed transformations are presented in a longer technical report [19].

We now detail how our formalism helps in maintaining the models up-to-date by detailling the process in the case of our three example modifications presented in Section 1.1.

1. A transformation equivalent or almost equivalent, whose transposition is trivial: the analyst renames the class `Item` into `Page`.
2. A transformation on `JCT`, that cannot be represented in `PADL`: the developer removes the first statement of the body of the `getListener` method within the `Listener` class.
3. A transformation that can be represented in both models, but whose transposition is non-trivial: the developer adds the compilation unit `Answerer.java`, containing the class `Answerer`, in the package `httpserver`

4.4.1 Trivial Transposition

The modification is made by the analyst and the representation of the transformation as an Hoare triple is:

$$\{E[\textit{httpserver.Item}], \neg E[\textit{httpserver.Page}]\}$$
Change attribute `name` in `class` *httpserver.Item* from *Item* to *Page*
$$\{E[\textit{httpserver.Page}], \neg E[\textit{httpserver.Item}]\}$$

The transposed transformation is the same, the only transposition we need is a detail of implementation such that we are able to apply it on the source code.

4.4.2 Transformation without Effect on the Design Model

The modfication is made by the developer and the representation of the transformation as an Hoare triple is:

$$\{E[\mathit{httpserver.Listener\#getListener(\#main\_block\#0}]\}$$
Remove `statement` n°*0* of `main block` of
`method` *getListener* in `class` *httpserver.Listener*
$$\{\neg E[\mathit{httpserver.Listener\#getListener(\#main\_block\#0}]\}$$

This transformation does not have any effect on the design model, therefore, it is transposed to $\mathbb{I}_{\texttt{PADL}}$.

4.4.3 Non-trivial Transposition

The modification is made by the developper and the representation of the transformation as an Hoare tripe is:

$$\{E[\mathit{IAnswerer.java}, E[\mathit{IAnswerer.java/IAnswerer}],$$
$$E[\mathit{httpserver}], \neg E[\mathit{httpserver/IAnswerer.java}]\}$$
Add `compilation unit` *IAnswerer.java* in `package` *httpserver*
$$\{E[\mathit{IAnswerer.java}], E[\mathit{IAnswerer.java/IAnswerer}],$$
$$E[\mathit{httpserver}, E[\mathit{httpserver/IAnswerer.java}]\}$$

Compilation units do not exist in `PADL`. Using the specification defined in Section 4.3.2, the transposition of this transformation is:

$$\{E[\mathit{IAnswerer}, E[\mathit{httpserver}], \neg E[\mathit{httpserver.IAnswerer}]\}$$
Add `compilation unit` *IAnswerer.java* in `package` *httpserver*
$$\{E[\mathit{IAnswerer}, E[\mathit{httpserver}], E[\mathit{httpserver.IAnswerer}]\}$$

The transposed transformation modifies the design model by adding the class to the package, which is exactly the result of adding the compilation unit to the package. This procedure is illustrated in Fig. 6.

5 Related Work

Most of related articles on traceability focus on notifying developers if traceability links are on the verge of destruction [25] or on the recovery of traceability after-hand [20]. For example, Reiss [25] provides a tool to define and check traceability-related constraints among different documents, (*e.g.*, an interface in the source code must appear in the UML diagram) and notify developers if a constraint is broken. Our approach aims to preserve traceability links by keeping the different documents up-to-date and, therefore, enforcing their consistency automaticaly.

Many of the work on traceability involve non structured [4] or annexe (*i.e.*, not direct representation of the system) [26] documents. For example, Rompaey and Demeyer [26] present different strategies based on static analysis to extract

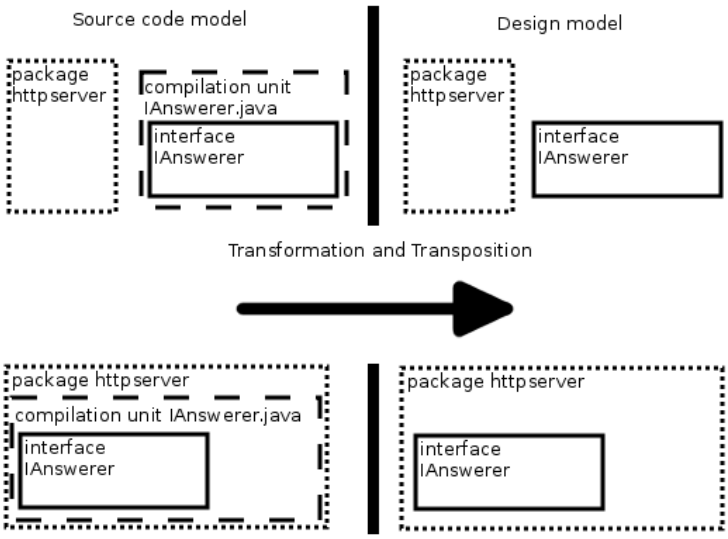

Figure 6: Non-trivail Transposition of `AddPackageCompilationUnit`.
Boxes in short dashes are packages;
Boxes in long dashes are compilation units;
Boxes in full line are interfaces.

traceability links between source code and test units. Our approach focuses on different structured representations of a system.

Transformations, as refactorings, were first studied by Opdyke [24] and Fowler [13]. Refactorings are coarse-grained and represent specific modifications of a system that does not change its behaviour. Transformations are fine-grained and can represent any modification of a system. Moreover, their work consists mostly in a catalog of refactorings and they never presented a formalism ground on and satisfying mathematical properties.

An usual representation of transformations is graph transformations [7, 21]. However, the properties required for the transformations we defined (*e.g.*, associativity) cannot be simply expressed in term of graph transformations. Moreover, both representations are not incompatible and Hoare triple can be seen only as an abstraction of graph transformations.

Abrial [2] noticed that formal methods could be hard to apply in industry. However, unlike formal methods, such as ASM [8], B [1], or VDM [18], our goal is not to describe a methodology but to use a mathematical formalism to guarantee properties used by other tools. Therefore, the end-users do not need to know the formalism details to use the final tools designed using our approach.

Oda and Saeki [23] focus on version control. They represent a model as a graph and define a set of operations as creation, deletion and update of model constituents to keep an historic of different versions of a model. However, Oda and Saeki do not present a way to transpose their operations to any other

system representation. A more advanced version of our formalism could be used to provide a structurally-aware, fine-grained version control, using properties similar to DARCS patch theory [10], such as commutativity.

Moha [22] represents transformations under many meta-models at the same time. However, this representation requires that we ensure the mathematical properties of our transformations for each meta-model under which transformations are defined. Moha representation also binds together the meta-models and requires an evolution of the transformations themselves when adding a new meta-model to our approach. Therefore, we prefer to define an explicit transposition from one meta-model to another one. Using our representation, we can define simpler transformations. Moreover, the modifications required to add a new meta-model are only the addition of transpositions between the new meta-model and the other ones we need to use.

The most similar work to ours is that by Diskin [11]. Both of our approaches include similar properties and elements. For example, Diskin's diagonal synchronization systems are used similarly to a pair of transpositions, both of these elements provide the ability to keep models up-to-date and consistent. Both also include similar properties, such as reversibility (denoted as *(Undo)* by Diskin), which is mandatory for all our transformations. Hippocraticness may be compared to our property of stability. However, Diskin's mathematical approach focuses on models as first-class entities and transformations as models-valued functions, whereas our approach use both models and transformations as first-class entities. Moreover, Diskin uses functions analysis, which is part of set theory as mathematical background ; we use group theory, which is a part of abstract algebra.

Briefly, our approach aims to preserve traceability automatically by keeping the documents of a software system up-to-date and consistent. Our formalism, focusing on structured and direct representations of the system, provides a mathematical formalization of transformations using abstract algebra. We can add or remove an arbitrary meta-model without any modification on the others or on the transformations thanks to a total separation of meta-models.

6 Conclusion and Future Work

The formalism presented in this paper uses meta-models to describe models representing the artifacts of software systems. The models are manipulated using transformations. We defined transformations as sets of primitive transformations that can be combined together with sequencing. Our formalism is grounded in mathematical theory, guaranteeing the required properties, including reversibility and associativity of transformations and coherence and stability of transpositions. Our implementation of transformations transpositions between meta-models automaticaly keeps up-to-date models and retrieves any version of different models of a software system.

Our formalism can be used in many other context and use-cases. For example, version control, language translation/migration, automated API upgrade

and patch generation. Some of these uses would require more meta-models and more complex implementation, but the formalism guarantees the basic required properties.

As future work, the formalism will be improved by specifying more advanced and complex operations on predicates (*e.g.*, include commutativity [10]) and or transformations (refactorings [13]) and their detection in a sequence of transformations.

Acknowledgments

We thanks Stéphane Vaucher for constructive criticism of an earlier version of this paper.

References

[1] Jean-Raymond Abrial. *The B-Book, assigning programs to meaning.* Cambridge University Press, 1996.

[2] Jean-Raymond Abrial. Formal methods in industry: achievements, problems, future. In Leon J. Osterweil, H. Dieter Rombach, and Mary Lou Soffa, editors, *ICSE*, pages 761–768. ACM Press, 2006. ISBN: 1-59593-375-1.

[3] Hervé Albin-Amiot. *Idiomes et Patterns Java : Application à la Synthèse de Code et à la Détection.* Thèse de doctorat, université de Nantes, février 2003.

[4] Giuliano Antoniol, Gerardo Canfora, Gerardo Casazza, Andrea De Lucia, and Ettore Merlo. Recovering traceability links between code and documentation. *TSE*, 28(10):970–983, Piscataway, NJ, USA, 2002. IEEE Press.

[5] Paulo E. S. Barbosa, Franklin Ramalho, Jorge C. A. de Figueiredo, and Antonio D. dos S. Junior. An extended MDA architecture for ensuring semantics-preserving transformations. In *SEW '08 Proceedings of the 2008 32nd Annual IEEE Software Engineering Workshop*, pages 33–42, 2008.

[6] Ira D. Baxter and Christopher W. Pidgeon. Software change through design maintenance. In M. J. Harrold and G. Visaggio, editors, *ICSM*, pages 250–259. IEEE Computer Society, 1997. ISBN: 0-8186-8013-X.

[7] Dorothea Blostein and Andy Schürr. Computing with graphs and graph transformations. *Softw., Pract. Exper.*, 29(3):197–217, 1999.

[8] Egon Börger and Robert F. Stärk. *Abstract State Machines. A Method for High-Level System Design and Analysis.* Springer, 2003.

[9] Jean Bézivin and Nicolas Ploquin. Tooling the MDA framework: A new software maintenance and evolution scheme proposal. In Richard Wiener, editor, *Journal of Object-Oriented Programming*, 14(12). SIGS Publications, December 2001.

[10] DARCS. Understanding darcs, patch theory. `http://en.wikibooks.org/wiki/Understanding_darcs/Patch_theory`, 2008.

[11] Zinovy Diskin. Algebraic models for bidirectional model synchronization. In Krzysztof Czarnecki, Ileana Ober, Jean-Michel Bruel, Axel Uhl, and Markus Völter, editors, *MoDELS*, volume 5301 of *Lecture Notes in Computer Science*, pages 21–36. Springer, 2008. ISBN: 978-3-540-87874-2.

[12] David S. Dummit and Richard M. Foote. *Abstract Algebra*. Wiley, third edition, 2004.

[13] Martin Fowler. *Refactoring: improving the design of existing code*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999. ISBN: 0-201-48567-2.

[14] Orlena C. Z. Gotel and Anthony C. W. Finkelstein. An analysis of the requirements traceability problem. In *First Internationnal Conference on Requirements Engineering*, pages 94–101. IEEE Computer Society, 1994.

[15] Yann-Gaël Guéhéneuc. *Un cadre pour la traçabilité des motifs de conception*. Thèse de doctorat, École des Mines de Nantes et Université de Nantes, juin 2003.

[16] Yann-Gaël Guéhéneuc and Giuliano Antoniol. Demima: A multi-layered framework for design pattern identification. *TSE*. IEEE Computer Society Press, September 2008. *Accepted for publication.*

[17] Charles A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, New York, NY, USA, 1969. ACM Press.

[18] Cliff B. Jones. *Systematic Software Development using VDM*. Prentice-Hall, 1990.

[19] Mathieu Lemoine and Yann-Gaël Guéhéneuc. A formalism for transformations traceability. Technical report RT1346, Université de Montréal, 2009.

[20] Andrea De Lucia, Massimiliano Di Penta, Rocco Oliveto, and Francesco Zurolo. Coconut: Code comprehension nurturant using traceability. In *ICSM*, pages 274–275. IEEE Computer Society, 2006. ISBN: 0-7695-2354-4.

[21] Tom Mens, Niels Van Eetvelde, Serge Demeyer, and Dirk Janssens. Formalizing refactorings with graph transformations. *Journal of Software Maintenance*, 17(4):247–276, 2005.

[22] Naouel Moha, Vincent Mahé, Olivier Barais, and Jean-Marc Jézéquel. Generic model refactorings. In Andy Schürr and Bran Selic, editors, *MoDELS*, volume 5795 of *Lecture Notes in Computer Science*, pages 628–643. Springer, 2009. ISBN: 978-3-642-04424-3.

[23] Takafumi Oda and Motoshi Saeki. Generative technique of version control systems for software diagrams. In Tibor Gyimóthy and Vaclav Rajlich, editors, *21st IEEE International Conference on Software Maintenance (ICSM 2005), 25-30 September 2005, Budapest, Hungary*, pages 515–524. IEEE Computer Society, 2005. ISBN: 0-7695-2368-4.

[24] William F. Opdyke. *Refactoring Object-Oriented Frameworks*. Ph.D. thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, 1997.

[25] Steven P. Reiss. Incremental maintenance of software artifacts. In Tibor Gyimóthy and Vaclav Rajlich, editors, *21st IEEE International Conference on Software Maintenance (ICSM 2005), 25-30 September 2005, Budapest, Hungary*, pages 113–122. IEEE Computer Society, 2005. ISBN: 0-7695-2368-4.

[26] Bart Van Rompaey and Serge Demeyer. Establishing traceability links between unit test cases and units under test. In Andreas Winter, Rudolf Ferenc, and Jens Knodel, editors, *CSMR*, pages 209–218. IEEE, 2009.

CHAPITRE 4

WATCHER, UN PLUGICIEL ECLIPSE

La génération d'une séquence de transformations à la main est fastidieuse et peu utile, ainsi nous avons développé un plugiciel Eclipse permettant d'extraire un liste de transformations à partir des modifications réalisées sur le code source par un développeur dans Eclipse tel que présenter dans la Figure 1.1.

4.1 Présentation du plugiciel

Watcher est un plugiciel Eclipse chargé au démarrage de l'IDE et s'intègrant à l'éditeur de code source Java. À chaque fois qu'un nouveau fichier de code source est ouvert, il en enregistre le contenu ainsi que le chemin vers le fichier (selon Eclipse) et le moment d'ouverture. À chaque modification, ajout ou suppression de texte dans le fichier, la modification est enregistrée avec le chemin vers le fichier sur lequel elle a lieu, ainsi que le moment de la modification. Les modifications sont communiquées par Eclipse en tant qu'ajout de texte ou suppression de texte d'une longueur précise à une position précise (au caractère près) dans le fichier. Elles sont sauvegardées dans deux fichiers (un contenant les fichiers de codes sources sauvegardés tels qu'à leur ouverture et un autre contenant la liste de leurs modifications).

4.2 Traitement des données

Nous utilisons les données générées par Watcher pour extraire une liste de transformations sur une instance de JCT, notre modèle pour le code source, correspondant aux fichiers de code source ouverts. Le traitement se décompose en 4 étapes :

1. Extraction des fichiers de code source ;
2. Conversion des fichiers de code sources extraits en instances sérialisées (*serialized*) de JCT ;

3. Filtrage des modifications textuelles pour qu'elles référencent les instances sérialisées de JCT appropriées ;

4. Conversion des modifications en transformations JCT concrètes.

Chaque étape est séparée des autres et enregistre les informations nécéssaires dans des fichiers externes afin de pouvoir les reprendre à l'étape suivante sans avoir à refaire les calculs. Les trois premières étapes consistent respectivement à :

1. Parcourir le fichier contenant toutes les sauvegardes de fichiers de code source ouverts et à recréer chaque fichier ;

2. Analyser les fichiers ainsi extraits avec l'implémentation de JCT afin d'obtenir une instance de JCT représentant chaque fichier. Ces instances de JCT sont sauvegardée ensuite dans des fichiers séparés. Les fichiers produisant des erreurs de compilation lors la création de l'instance JCT sont ignorés et retirés de la liste ;

3. Parcourir le fichier contenant l'ensemble des modifications du code source en remplaçant les informations concernant le fichier source qui a été modifié par les informations concernant le fichier contenant l'instance de JCT appropriée. Si l'instance de JCT n'existe pas, car une erreur est survenue à l'étape précédente, la modification est retirée de la liste.

La dernière étape est plus complexe. Elle consiste à extraire des transformations à partir de modifications textuelles. Chaque modèle pouvant être représenté comme un graphe, ce problème est assimilable au problème de correspondance de graphes (*graph matching*) [5], encore ouvert dans le domaine de la recherche bien que des solutions intéressantes existent.

L'implémentation développée dans ce mémoire sert seulement de validation sommaire et permet la détection de transformations sur les noms de classes, méthodes, champs (*fields*) et paramètres de méthodes. La position de chaque modification dans le fichier source permet de déterminer précisément sur quel élément de l'instance de JCT elle est effectuée. Une analyse syntaxique reposant sur les caractères réservés du

langage permet de déterminer si l'emplacement de la modification correspond au nom de l'élément modifié. Si c'est le cas, une transformation est générée pour représenter le changement de nom. Les autres transformations sont ignorées.

Bien que sommaire notre implémentation est une preuve de concept montrant qu'il est possible d'extraire automatiquement les transformations à partir de modifications textuelles. Nous envisageons dans un travail futur d'extendre Watcher pour supporter un éventail complet de transformations.

CHAPITRE 5

GÉNÉRATEUR DE CODE

Notre formalisme n'étant pas restreint à l'utilisation de JCT et PADL, il est possible de développer ou d'intégrer d'autres méta-modèles à notre implémentation. Cependant, générer la liste des transformations peut-être long et fastidieux. C'est pourquoi nous avons développé un générateur de code permettant la génération ou l'intégration de nouveaux méta-modèles.

5.1 Présentation du générateur de code

Le générateur de code remplit deux rôles :

1. Il permet de générer de nouveaux méta-modèles complexes à partir d'une version allégées n'en représentant que la structure.

2. Il permet de générer l'ensemble des classes de transformations primitives possibles sur un méta-modèle (nouveau ou pré-existant) à partir de cette même structure.

Le générateur de code génère du code source Java documenté avec la syntaxe de `javadoc`. Le code source est séparé en deux parties, interfaces et implémentation, générables séparément, ainsi que certains utilitaires, telles qu'une interface de Visiteur (*Visitor*) ou une Fabrique (*Abstract Factory*) [16], qui permettent le développement et l'utilisation d'implémentations alternatives. Notre générateur possède également des mécanismes d'extension permettant de l'adapter à la plupart des conventions de nommage et d'apposer une licence ou une notice de droit d'auteur arbitraire sur chacun des fichiers qu'il génère, ce qui permet d'utiliser le code source généré dans de nombreux contextes.

5.1.1 Génération de nouveaux méta-modèles

La génération de nouveaux méta-modèles se fait à partir d'une description de la structure du méta-modèle décrite selon une syntaxe à base de S-expressions (expressions

parenthésées).

Si des méthodes non-triviales doivent être incluses dans l'implémentation du méta-modèle, elles peuvent être intégrées directement dans la description de la structure ou écrites dans un fichier additionnel contenant le code source à intégré dans le code généré.

Par exemple, pour JCT, la spécification des `imports` est présentée dans la Figure 5.1. La liste des informations inclut notamment :

L 1 : Le nom du constituant ;

L 3,4 : L'interface et les classes dont dérivent les interface et classe représentant le constituent ;

L 8 : La date de création et le nom de l'auteur ;

L 9-16 : Les commentaires décrivant l'interface représentant le constituent ;

L 17-20 : La liste des champs du constituant avec leur type, ainsi que diverses informations, par exemple le drapeau `:interface-only-accessor` précise que l'implémentation des accesseurs ne doit pas être générée. Il peut être utilisé dans le cas de fonctions non triviales qui sont spécifiés dans le fichier additionnel ;

L 21-28 : La liste des méthodes à ajouter dans les interfaces et classes représentant le constituant.

L 29 : Le chemin vers le fichier séparé à ajouter à la classe représentant le constituent si nécessaire.

Cette description permet de créer une structure de données contenant les informations nécessaires à chaque constituant. Ces structures sont ensuite parcourues l'une après l'autre pour générer les diverses interfaces et classes Java de l'implémentation.

5.1.2 Génération des transformations possibles pour un méta-modèle

La génération des transformations pour un méta-modèle se fait grâce à un parcours de la description du méta-modèle. Les transformations sont générées à partir de cette

```
1  (create-mm-constituent-w/impl-file-helper
2     "Import" ()
3     #T('jct "Source Code Part")
4     `(,#T('jct "Source Code Part"))
5     ()
6     ()
7     '("java.io.IOException")
8     "2009-01-08" "Mathieu Lemoine"
9     '("This class represents an import"
10      "Imports respect Java restrictions:"
11      "<ul>"
12      "<li>if an import is on-demand and static, the imported element
           must be a top level class.</li>"
13      "<li>if an import is on-demand but not static, the imported
           element must be a package</li>"
14      "<li>if an import is static but not on-demand, the imported
           element must be a static class member</li>"
15      "<li>if an import is neither static nor on-demand, the imported
           element must be a class</li>"
16      "</ul>")
17    (list
18      (create-mm-field "imported element" #T('jct "importable")
           :interface-only-accessor)
19      (create-mm-field "is static"        #T(nil "boolean")
           :interface-only-accessor)
20      (create-mm-field "is on demand"     #T(nil "boolean")
           :interface-only-accessor))
21    (list
22      (create-whole-java-method
23         (() nil "JCTImport" '(("IJCTRootNode" "aRootNode")
24                               ("IJCTImportable" "importedElement")
25                               ("boolean" "isStatic")
26                               ("boolean" "isOnDemand")))
27         :abstract
28         "Sole Constructor")
29    "jct.kernel.impl.JCTImport.java")
```

Figure 5.1 – Spécification des `imports` de JCT dans le générateur de code

description et appellent directement l'API du méta-modèle. Il n'est pas nécessaire pour le méta-modèle d'avoir été généré par le générateur de code (*e.g.*, cette technique a été utilisée pour générer les transformations sur PADL).

Pour chaque constituant, une transformation pour la création et la destruction du constituant est créée. Pour chaque champ de chaque constituant, deux choix sont possibles :

1. Si le champ est une liste ou un ensemble de valeurs, des transformations permettant l'ajout et la suppression de valeurs dans la liste sont créées.

2. Si le champs est une valeur atomique, une transformation permettant de modifier la valeur du champ est créée.

CHAPITRE 6

CONCLUSION ET TRAVAUX FUTURS

6.1 Conclusion

Les problèmes de traçabilité et de maintien à jour des divers documents composant un système logiciel sont parmi les pricipaux problèmes auquels doivent faire face les développeurs dans le cadre du développement et de la maintenance de tels systèmes dans l'industrie. Afin d'éviter que certains documents ne deviennent obsolètes ou incohérents, nous avons proposé une approche reposant sur les formalismes mathématiques des triplets de Hoare, issus de l'informatique théorique, et de la théorie des groupes et des homomorphismes de groupes, issus de l'algèbre abstraite.

Ces formalismes permettent la manipulation des composants de plusieurs représentations d'un système logiciel, en garantissant des propriétés telles que la reversibilité ou l'associativité sur les modifications appliquées. Ces propriétés permettent à leur tour des manipulations automatiques de ces transformations afin de les reporter sur d'autres représentations du logiciel que celle sur laquelle elles sont concrètement appliquée. Ces transpositions permettent ainsi de conserver les documents à jour et cohérents.

Nous avons illustré notre approche grâce à une implémentation et un exemple utilisant cette implémentation. Ces résultats font l'objet d'une soumission d'un article pour le *Journal of Software Maintenance and Evolution: Research and Practice* .

Afin de faciliter l'application de notre approche dans un contexte réel de développement, nous avons développé un plugiciel non-intrusif pour Eclipse, un des environnements de développement en Java les plus utilisés. Ce plugiciel permet d'enregistrer automatiquement les modifications textuelles du code source effectuées par les développeurs. Nous avons également développé un analyseur syntaxique des données enregistrées, permettant d'extraire automatiquement des transformations sur notre implémentation d'AST (JCT) à partir des modifications textuelles.

Enfin, pour faciliter l'application de notre formalisme à d'autre méta-modèles, nous

avons développé un générateur de code permettant de générer rapidement de nouveaux méta-modèles et d'intégrer des méta-modèles existants en générant les transformations pour ces méta-modèles.

6.2 Travaux Futurs

Ce mémoire présente une nouvelle approche au problème de cohérence et de maintien à jour automatique de plusieurs modèles du même système logiciel. Cette solution représente une base ouvrant la voie à de nouveaux développements.

6.2.1 Le formalisme

Concernant le formalisme lui-même, certaines propriétés supplémentaires pourraient être intégrées au besoin, par exemple, la commutativité des transformations, la possibilité de permuter deux transformations dans une séquence, bien qu'elle ne soit pas triviale pour toutes les transformation (*e.g.*, permuter la création et le renommage d'un objet reviendrait à renommer un objet qui n'est pas encore créé, la sémantique associée à cette permutation n'est pas triviale). Il est certainement possible de mettre en place un formalisme similaire à celui utilisé dans le gestionnaire de versions DARCS [10]. Ce formalisme permet de modifier (au moins) une des deux transformations permutées afin de prendre en compte les cas problématiques (*e.g.*, un constituant change de nom) ou de refuser la permutation si le cas n'a pas de sémantique claire (comme le cas de création et renommage mentionné précédemment). L'ajout de nouvelles propriétés permettrait de faire évoluer le formalisme mathématique utiliser et d'utiliser de nouveaux outils mathématiques permettant des analyses plus complexes sur les transformations (la modélisation et détection de réusinages ou la mise à jour automatique d'API, par exemple).

Le formalisme que nous avons proposé n'est pas cantonné à notre problème de traçabilité, il pourrait être appliqué à d'autres voies de recherches. La représentation et la formalisation des réusinages, comme indiqué ci-dessus, est une application qui pourrait être intégrée.

Un autre exemple est la traduction de programmes écrits dans différents languages de programmation. Toute transformation étant inversible par définition, l'inverse de la transformation supprimant l'ensemble d'un système logiciel revient à recréer le système à partir d'un ensemble de transformations. En utilisant cette propriéte et pour des langages avec des modèles similaires ou en utilisant des transpositions spécialisées, il pourrait être possible d'obtenir un outil de traduction de langage sommaire. Ce traducteur, se basant uniquement sur les propriétés syntaxiques du langage, et procédant donc par restructuration, posséderait certainement des caractéristiques similaire à Cappucino, présenté par Buddrus et Schödel [9]. Cependant les deux versions du logiciels pourraient être maintenues à jour grâce à l'application du formalisme.

6.2.2 L'implémentation

JCT repose sur le langage Java. Certains points particuliers du langage ne sont cependant pas supportés complétement, les *generics* ou les `enums` par exemple. Intégrer les derniers éléments du langage à JCT permettrait de mieux supporter les système logiciels développés en utilisant ces fonctionnalités.

6.2.3 Le plugiciel

La quatrième étape du traitement des données produites par Watcher que nous avons développé n'est qu'une validation sommaire, le concept est réaliste et fonctionne. La séparation des étapes du traitement rend très facile de l'intégration d'une implémentation plus complète ou plus efficace pour remplace l'implémentation exemple actuelle. Cependant, la mise en place d'une telle implémentation demanderait beaucoup de temps de développement. Un amélioration possible serait l'intégration d'algorithmes récents et performants, tels que [14, 26, 33], pour la correspondance de graphe ou la compilation incrémentielle.

6.2.4 Le générateur de code

Bien qu'il soit simple à utiliser, certaines spécifications de convention de nommage ou de développement peu usitées ne sont pas triviales à insérer dans le générateur de code. Une refonte de l'API du générateur de code, de meilleures simplifications et documentations des mécanismes d'extension et la création d'un DSL adéquat (qui pourrait ensuite être formalisée sous la forme d'un méta-méta-modèle) pour la spécification des méta-modèles permettraient une intégration plus aisée de nouveaux méta-modèles ou de méta-modèles existants à notre formalisme ainsi qu'une meilleure séparation entre la spécification des méta-modèles et le fonctionnement interne du générateur.

BIBLIOGRAPHIE

[1] Jean-Raymond Abrial. *The B-Book, assigning programs to meaning*. Cambridge University Press, 1996.

[2] Jean-Raymond Abrial. Formal methods in industry: achievements, problems, future. In Leon J. Osterweil, H. Dieter Rombach, and Mary Lou Soffa, editors, *ICSE*, pages 761–768. ACM Press, 2006. ISBN: 1-59593-375-1.

[3] Hervé Albin-Amiot. *Idiomes et Patterns Java : Application à la Synthèse de Code et à la Détection*. Thèse de doctorat, université de Nantes, février 2003.

[4] Giuliano Antoniol, Gerardo Canfora, Gerardo Casazza, Andrea De Lucia, and Ettore Merlo. Recovering traceability links between code and documentation. *TSE*, 28(10):970–983, Piscataway, NJ, USA, 2002. IEEE Press.

[5] Maria Axenovich, André E. Kézdy, and Ryan Martin. On the editing distance of graphs. *Journal of Graph Theory*, 58(2):123–138, 2008.

[6] Ira D. Baxter and Christopher W. Pidgeon. Software change through design maintenance. In M. J. Harrold and G. Visaggio, editors, *ICSM*, pages 250–259. IEEE Computer Society, 1997. ISBN: 0-8186-8013-X.

[7] Dorothea Blostein and Andy Schürr. Computing with graphs and graph transformations. *Softw., Pract. Exper.*, 29(3):197–217, 1999.

[8] Egon Börger and Robert F. Stärk. *Abstract State Machines. A Method for High-Level System Design and Analysis*. Springer, 2003.

[9] Frank Buddrus and Jörg Schödel. Cappuccino - A C++ to Java translator. In *SAC*, pages 660–665. ACM Press, 1998.

[10] DARCS. Understanding darcs, patch theory. `http://en.wikibooks.org/wiki/Understanding_darcs/Patch_theory`, 2008.

[11] DARCS. Darcs homepage, updated september 20th, 2009. `http://www.darcs.net`, 2009.

[12] Office Québécois de la Langue Française. Grand dictionnaire — *refactoring*. `http://www.granddictionnaire.com/`, 2009. [En ligne, Consulté le 09-Déc-2009, Entrée *refactoring*].

[13] Zinovy Diskin. Algebraic models for bidirectional model synchronization. In Krzysztof Czarnecki, Ileana Ober, Jean-Michel Bruel, Axel Uhl, and Markus Völter, editors, *MoDELS*, volume 5301 of *Lecture Notes in Computer Science*, pages 21–36. Springer, 2008. ISBN: 978-3-540-87874-2.

[14] Beat Fluri, Michael Würsch, Martin Pinzger, and Harald Gall. Change distilling: Tree differencing for fine-grained source code change extraction. *IEEE Trans. Software Eng.*, 33(11):725–743, 2007.

[15] Martin Fowler. *Refactoring: improving the design of existing code*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999. ISBN: 0-201-48567-2.

[16] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Abstraction and Reuse of Object-Oriented Design*. Addison-Wesley, Boston, MA, January 1995. ISBN: 0201633612.

[17] Orlena C. Z. Gotel and Anthony C. W. Finkelstein. An analysis of the requirements traceability problem. In *First Internationnal Conference on Requirements Engineering*, pages 94–101. IEEE Computer Society, 1994.

[18] Yann-Gaël Guéhéneuc. *Un cadre pour la traçabilité des motifs de conception*. Thèse de doctorat, École des Mines de Nantes et Université de Nantes, juin 2003.

[19] Yann-Gaël Guéhéneuc and Giuliano Antoniol. Demima: A multi-layered framework for design pattern identification. *TSE*. IEEE Computer Society Press, September 2008. *Accepted for publication.*

[20] Ahmed E. Hassan. The road ahead for mining software repositories. In *Proceedings of the Future of Software Maintenance (FoSM) at the 24th IEEE International Conference on Software Maintenance (ICSM)*, 2008.

[21] Lile Hattori, Gilson dos Santos Jr, Fernando Cardoso, and Marcus Sampaio. Mining software repositories for software change impact analysis: a case study. In *Proceedings of the 23rd Brazilian symposium on Databases*, pages 210–223, 2008.

[22] Charles A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, New York, NY, USA, 1969. ACM Press.

[23] Mohammad-Amin Jashki, Reza Zafarani, and Ebrahim Bagheri. Towards a more efficient static software change impact analysis method. In *Proceedings of the 8th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 84–90, 2008.

[24] Cliff B. Jones. *Systematic Software Development using VDM*. Prentice-Hall, 1990.

[25] Andrea De Lucia, Massimiliano Di Penta, Rocco Oliveto, and Francesco Zurolo. Coconut: Code comprehension nurturant using traceability. In *ICSM*, pages 274–275. IEEE Computer Society, 2006. ISBN: 0-7695-2354-4.

[26] Sergey Melnik, Hector Garcia-Molina, and Erhard Rahm. Similarity flooding: A versatile graph matching algorithm and its application to schema matching. In *ICDE*, pages 117–128. IEEE Computer Society, 2002. ISBN: 0-7695-1531-2.

[27] Tom Mens, Niels Van Eetvelde, Serge Demeyer, and Dirk Janssens. Formalizing refactorings with graph transformations. *Journal of Software Maintenance*, 17(4):247–276, 2005.

[28] Naouel Moha, Vincent Mahé, Olivier Barais, and Jean-Marc Jézéquel. Generic model refactorings. In Andy Schürr and Bran Selic, editors, *MoDELS*, volume 5795 of *Lecture Notes in Computer Science*, pages 628–643. Springer, 2009. ISBN: 978-3-642-04424-3.

[29] Takafumi Oda and Motoshi Saeki. Generative technique of version control systems for software diagrams. In Tibor Gyimóthy and Vaclav Rajlich, editors, *21st IEEE International Conference on Software Maintenance (ICSM 2005), 25-30 September 2005, Budapest, Hungary*, pages 515–524. IEEE Computer Society, 2005. ISBN: 0-7695-2368-4.

[30] William F. Opdyke. *Refactoring Object-Oriented Frameworks*. Ph.D. thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, 1997.

[31] Steven P. Reiss. Incremental maintenance of software artifacts. In Tibor Gyimóthy and Vaclav Rajlich, editors, *21st IEEE International Conference on Software Maintenance (ICSM 2005), 25-30 September 2005, Budapest, Hungary*, pages 113–122. IEEE Computer Society, 2005. ISBN: 0-7695-2368-4.

[32] Bart Van Rompaey and Serge Demeyer. Establishing traceability links between unit test cases and units under test. In Andreas Winter, Rudolf Ferenc, and Jens Knodel, editors, *CSMR*, pages 209–218. IEEE, 2009.

[33] Sven Wenzel, Hermann Hutter, and Udo Kelter. Tracing model elements. In *ICSM*, pages 104–113. IEEE Computer Society, 2007.

[34] Jianjun Zhao, Hongji Yang, Liming Xiang, and Baowen Xu. Change impact analysis to support architectural evolution. *Journal of Software Maintenance: Research and Practice*, 14(5):317–333, 2002.

[35] Thomas Zimmermann. Mining workspace updates in cvs. In *Proceedings of the Fourth International Workshop on Mining Software Repositories*, May 2007.

[36] Thomas Zimmermann, Peter Weißgerber, Stephan Diehl, and Andreas Zeller. Mining version histories to guide software changes. *IEEE Transactions on Software Engineering*, 31(6):429–445, June 2005.