
Introduction du test dans la modélisation par aspects

Jacques Klein¹ — Benoit Baudry¹ — Olivier Barais¹ — Andrew Jackson²

¹ IRISA/INRIA Rennes – Université de Rennes 1
Campus Universitaire de Beaulieu
F-35042 Rennes Cedex, France
{jacques.klein, benoit.baudry, olivier.barais}@irisa.fr

² Distributed Systems Group, Computer Science Department,
Trinity College Dublin,
Dublin 2, Ireland
Andrew.Jackson@cs.tcd.ie

RÉSUMÉ. Dans un processus d'ingénierie dirigée par les modèles (IDM), les modèles représentent des vues sur le système et les transformations de modèles spécifient les relations entre ces vues. La modélisation par aspects (MPA) est une approche particulière de l'IDM qui permet de modéliser chaque préoccupation séparément. Les transformations de modèles permettent d'automatiser certaines étapes du développement comme les différentes compositions nécessaires à l'obtention du modèle du système. Mais lorsqu'une erreur est introduite dans un modèle, elle est propagée aux étapes suivantes et peut alors être très difficile à détecter et localiser. Les modèles de préoccupations étant plus compacts qu'un modèle complet, ils permettent de réduire l'espace pour les erreurs. Cependant, pour limiter le risque de propagation des erreurs, il est nécessaire d'avoir des techniques pour détecter les erreurs dans les modèles de préoccupations. Dans ce papier nous introduisons KerTheme comme une approche pour la MPA qui permet le test des modèles de préoccupations. KerTheme définit deux vues pour chaque préoccupation : un modèle exécutable et un modèle du comportement attendu. Le processus de test pour une préoccupation consiste alors à vérifier la cohérence entre la trace obtenue en exécutant le modèle et le comportement attendu.

ABSTRACT. In Model-Driven Software Development (MDS), models represent views of software at different levels of abstraction and transformations specify refinements from one level of abstraction to another. Aspect Oriented Modelling (AOM) complements MDS by extending the decomposition capabilities with separation of concerns at the same level of abstraction. Using transformations in software development has many benefits, but when an error is introduced into a model, it is propagated to later refinements. Such propagation makes it harder to trace errors to their source and consequently more difficult to correct. Concern models reduce the scope for error propagation as errors are localised. However, to ensure that errors are not propagated between concern models at different levels of abstraction, a technique for detecting errors in concern models is required. In this paper we present KerTheme, an approach to supporting error detection in AO models through

testing. In KerTheme, testing is supported by defining two views of a concern: an executable model and a model of its expected behaviour. The testing process consists of checking the consistency between a trace from executing the concern model and the expected behaviour of the concern.

MOTS-CLÉS : Test, Modélisation par aspects, Theme, Composition de modèles

KEYWORDS: Model composition, Aspect Oriented Modeling, Test, Theme

1. Introduction

Dans l'Ingénierie Dirigée par les Modèles (IDM), les modèles représentent les vues des systèmes logiciels à différents niveaux d'abstraction, et les transformations de ces modèles spécifient des raffinements d'un de ces niveaux d'abstraction vers un autre (Mukerji *et al.*, 2003). Un des bénéfices majeurs de l'IDM est qu'il augmente le niveau d'abstraction dans un développement logiciel (Lewis *et al.*, 2005). Ceci est réalisé à travers une séparation verticale du domaine logique et une mise en oeuvre des technologies dans différents modèles, qui partitionnent les décisions qui doivent être considérées à différentes phases du cycle de développement logiciel. Cette séparation permet de localiser où les décisions sont prises, et par conséquent, où les erreurs peuvent se produire. Une erreur apparaît quand, par exemple, les comportements définis dans les modèles ne sont pas conformes aux exigences. Lorsqu'une erreur se produit dans un modèle, elle est propagée à travers les transformations si elle n'est pas résolue immédiatement. La propagation d'erreur augmente la complexité liée à la localisation de la source d'erreur et par conséquent la difficulté liée à la correction de l'erreur.

La modélisation par aspects (MPA) est un domaine particulier de l'IDM qui supporte la décomposition de préoccupations dans des modèles séparés à un même niveau d'abstraction. Les préoccupations sont des "éléments" d'intérêt qui nécessitent d'être traités au cours du développement d'un système logiciel. Cette séparation horizontale étend la séparation verticale de l'IDM¹ en partitionnant davantage les décisions qui doivent être prises à chaque niveau d'abstraction. Un des avantages attendus de la définition des préoccupations dans différents modules est une meilleure maintenabilité des modèles. Les erreurs peuvent être "remontées" à travers différents niveaux d'abstraction puisque les modifications éventuelles sont localisées dans les préoccupations. Pour s'assurer que les erreurs ne sont pas propagées entre les modèles des préoccupations à différents niveaux d'abstraction, une technique pour détecter les erreurs dans les préoccupations modélisées est nécessaire. La décomposition des préoccupations à un niveau de modélisation est facilitée par l'existence de tisseur de préoccupations, pour produire le modèle complet d'un système logiciel. Le tissage est fondé sur des spécifications qui décrivent comment les préoccupations modélisées doivent être composées. Une mauvaise spécification d'un tissage peut introduire des erreurs dans les modèles composés. Pour détecter ces erreurs, le test est une approche pragmatique pour valider les différents modèles des préoccupations et les modèles composés.

IEEE décrit le test comme : "une activité dans laquelle un système ... est exécuté sous des conditions spécifiées, dont les résultats sont observés ou enregistrés et où une évaluation est faite de certains aspects du système" (IEEE, New York, September 2006).

1. exemple de séparation verticale : La transformation PIM vers PSM du MDA de l'OMG

Une interprétation plus abstraite de cette définition est que le test consiste en la validation de la cohérence entre deux vues d'un même système. Pour détecter des erreurs dans des modèles comportementaux, nous avons besoin de considérer deux vues de ces comportements. Pour tester des modèles par aspects, cela signifie que nous avons besoin de deux vues des comportements des préoccupations. Une vue qui décrit précisément comment le comportement des préoccupations doit être exécuté, et une autre qui décrit le comportement attendu de l'exécution. De plus, dans le contexte spécifique de la MPA, certains comportements ne peuvent pas être testés de façon isolée, car ils doivent, au préalable, être tissés avec les autres comportements pour que la phase de test puisse avoir lieu. Dans ce cas, des mécanismes efficaces de traçabilités sont nécessaires pour retrouver la source de l'erreur, soit dans une préoccupation erronée, soit dans spécification de tissage erronée.

Dans ce papier, nous proposons un environnement pour MPA, appelé KerTheme, qui offre les caractéristiques requises pour le test, et qui est développé dans la plateforme Kermeta (Muller *et al.*, 2005, Fleurey, 2006). KerTheme adapte Theme/UML (Clarke *et al.*, 2005) qui propose une décomposition symétrique des modèles. KerTheme définit des modules appelés kerThemes. Un kerTheme contient un scénario de haut niveau qui décrit le comportement attendu d'une préoccupation, ainsi qu'un diagramme de classes exécutable qui définit précisément le comportement exécuté. Pour ces deux vues, nous définissons des mécanismes de spécification de composition et des opérateurs de composition. Le processus de test, qui consiste à vérifier la cohérence entre les deux vues, est fondé sur l'analyse des traces d'exécution. Quand nous exécutons le diagramme de classes exécutable, nous sommes capables de construire une trace d'exécution, pour laquelle il est possible de vérifier sa présence dans les comportements définis par le scénario.

Cet article suit le plan décrit ci-après. La section 2 rappelle les idées de l'approche Theme qui sert de fondement à cette étude. La section 3 présente, à l'aide d'un exemple, le *framework* KerTheme permettant la modélisation par aspects d'un système et fournissant les caractéristiques requises pour le test. La section 4 présente les opérateurs de composition et de tissage qu'il a fallu développer pour fournir un environnement de modélisation par aspects. La section 5 présente le processus de test lié à la composition de kerThemes. La section 6 détaille les travaux connexes à ce travail. Finalement, la dernière section apporte une critique sur ce travail et met en évidence les travaux futurs à effectuer pour une prise en compte de la problématique de test dans les phases de modélisation d'un système par aspects.

2. Rappel sur Theme

Avant de décrire notre approche de test, notons que KerTheme est en partie fondé sur Theme (Clarke *et al.*, 2005) qui est une approche de la MPA supportant la décomposition d'un système en des modèles de préoccupations, appelés themes. Les préoccupations sont identifiées en utilisant la méthode Theme/Doc (Clarke *et al.*, 2005), et elles sont modélisées en themes en utilisant Theme/UML. Il existe deux types de themes : les themes de base et les themes d'aspect. Un theme et son type sont identifiés à travers l'analyse des exigences et l'identification des comportements qu'ils entrelacent. Un theme d'aspect est utilisé pour concevoir des préoccupations contenant des comportements transversaux. Les autres comportements sont identifiés comme des themes de base.

Un theme contient un diagramme de classe qui décrit la structure de la préoccupation modélisée. Un theme contient également un scénario qui décrit le comportement de la préoccupation modélisée.

3. KerTheme à travers un exemple

KerTheme vise à fournir un canevas de modélisation par aspects prenant en charge le test dès la phase de modélisation d'un système. Ainsi, à la différence d'un theme, dans un kerTheme, les scénarios ne sont pas utilisés pour décrire le comportement du système, mais pour décrire le comportement attendu du système. Le système est modélisé à l'aide d'un diagramme de classe exécutable.

Pour mieux comprendre KerTheme, nous allons présenter un exemple de système de ventes aux enchères. Bien qu'il y ait plusieurs préoccupations d'importances dans un tel système, nous allons uniquement nous intéresser à trois de ces préoccupations : *interface utilisateur*, *inscription* et *persistance*. *Interface utilisateur* est une préoccupation de base représentant le comportement entre un utilisateur et le système à travers une *vue* et un *contrôleur*, lors d'une inscription de l'utilisateur au système. *Inscription* est également une préoccupation de base qui représente le modèle métier d'une inscription d'un utilisateur dans le système. *Persistance* est une préoccupation d'aspect, c'est-à-dire transverse aux préoccupations de base, qui permet de sauvegarder des informations telles que celles fournies lors de l'inscription.

La représentation détaillée de ces préoccupations est décrite par les kerThemes des figures 1, 2 et 3. Chaque kerTheme comporte un diagramme de classe exécutable (même si sur les figures on ne voit que de simples diagrammes de classe) et un scénario.

Les scénarios utilisés dans KerTheme sont des Diagrammes de Séquence (DS) d'UML 2.0 restreint à des opérateurs de séquence, d'alternative et d'itération. Avec cette restriction, on peut considérer que les DSs ont la même sémantique que celle des Message Sequence Charts (MSCs) (ITU, 1996), et les résultats sur le tissage de scénarios présentés dans (Klein *et al.*, n.d., Klein *et al.*, 2006) peuvent alors être appliqués aux DSs.

Lorsqu'un kerTheme est une préoccupation de base, le scénario utilisé est un DS habituel, comme sur les figures 1 et 2. Le scénario de la figure 1 décrit les interactions entre l'utilisateur et le système à travers une *vue* et un *contrôleur* lors d'une inscription. Une validation des données nécessaire à l'inscription est également effectuée. Cette validation permet, par exemple, de vérifier qu'une adresse électronique est valide. Le scénario de la figure 2 décrit l'inscription d'un utilisateur sur le système. Le système (*register*) vérifie que l'utilisateur n'est pas déjà inscrit, et si ce n'est pas le cas, l'utilisateur est ajouté à la liste des inscrits. A la fin du scénario, la *vue* associée à l'utilisateur est mise à jour.

Lorsqu'un kerTheme est une préoccupation d'aspect, le scénario est en réalité un aspect comportemental, similaire à ceux définis dans (Klein *et al.*, n.d., Klein *et al.*, 2006). Un aspect comportementale est composé d'un scénario représentant l'expression de coupe, c'est-à-dire représentant le comportement à détecter, et un scénario représentant l'*advice*, c'est à dire le nouveau comportement aux endroits préalablement détectés. La figure 3 présente l'aspect comportemental lié à la préoccupation *Persistance*. A chaque

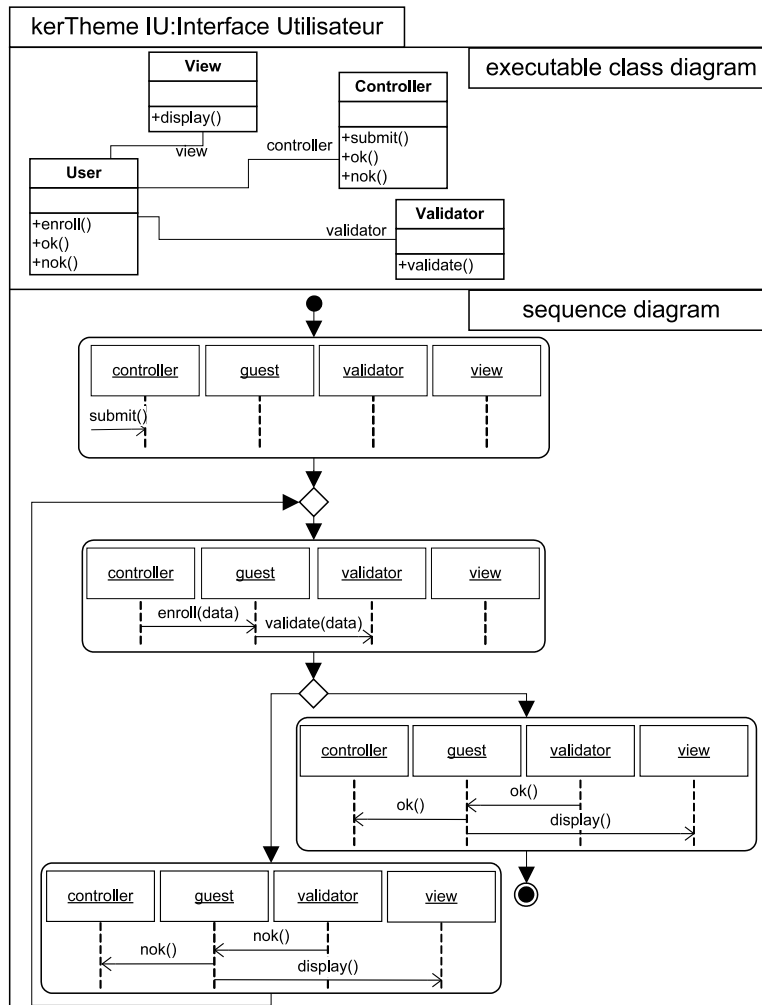


Figure 1. *kerTheme de base Interface Utilisateur*

fois qu'il y a un échange de messages entre un utilisateur et le système représenté par *register*, cet échange est sauvegardé dans une base de données.

Un *kerTheme* contient également un diagramme de classe exécutable qui permet d'exécuter les comportements associés aux préoccupations. Les trois figures 1, 2 et 3 présentent une vue des diagrammes de classe exécutable. Sur ces vues, seuls sont représentés les diagrammes de classe, l'exécutabilité est insufflée à ces classes en ajoutant du comportement dans les méthodes énumérées dans les diagrammes de classe. Pour ajouter ce comportement, nous utilisons Kermeta qui propose un modèle de classe fondée sur EMOF 2.0 et un langage d'action simple fondé sur le paradigme de la programmation par

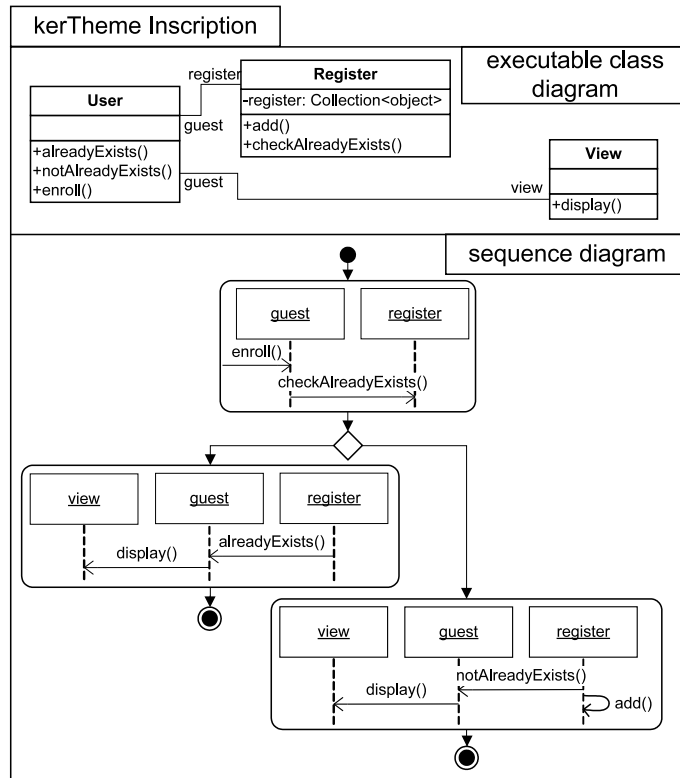


Figure 2. kerTheme de base Inscription

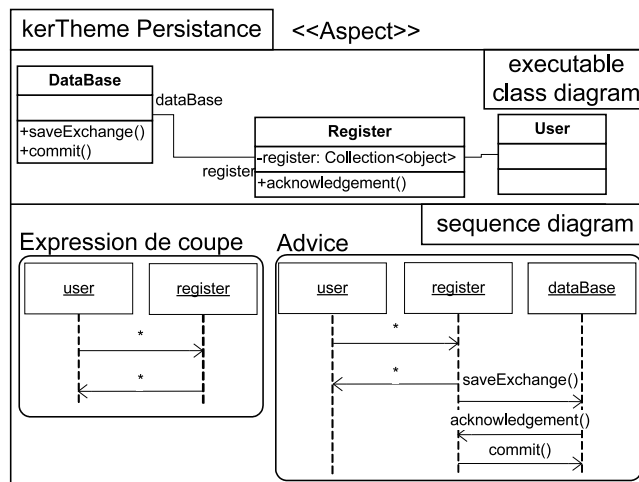


Figure 3. kerTheme d'aspect Persistence

objets. Le choix de Kermeta est discuté plus en détail en section 4.3. La figure 4 représente le code Kermeta relatif aux trois kerThemes étudiés.

| Interface Utilisateur | Inscription |
|--|--|
| <pre> class View{ reference controller : Controller operation display(message : String): Void is do end } class Controller{ reference user : User operation submit(data : Data) : Void is do user.enrol(data : Data) end operation ok() : Void is do end operation nok() : Void is do end } class User{ reference validator : Validator reference view : View reference controller: Controller operation enrol(data : Data) : Void is do validator.validate(data : Data) end operation ok() : Void is do controller.ok view.display(void) end operation nok() : Void is do controller.nok view.display(void) end } class Validator{ reference user : User operation validate(data: Data) : Boolean is do if data.email.isValid() then user.ok else user.nok end } class Data{ attribute email : Email } </pre> | <pre> class User{ reference register : Register#user reference view : View operation enrol(data Data) : Void is do register.check(data Data) end operation notAlreadyExists() : Void is do register.addToRegister(data) end operation alreadyExists() : Void is do view.display("already exists") end } class Register{ reference user : User#register attribute register : Set<Object> operation check(data : Data) : Void is do if register.exists(data.email) then user.alreadyExists() else user.notAlreadyExists() end end operation addToRegister(data Data) : Void is do register.add end } class View{ operation display(message : String) Void is do end } class Data{ attribute email : Email } </pre> |
| | <pre> Persistence class Database{ reference register :Register operation save(): Void is do register.ack() end operation commit() : Void is do end } class Register { reference database :Database operation ack() : Void is do database.commit end operation post(data: Data) : Void is do database.save(data) end } class User{} class Data{} </pre> |

Figure 4. Code Kermeta des trois kerThemes

4. Composition des kerThemes

Pour obtenir une description complète d'un système, nous devons être capables de composer les différentes préoccupations, c'est-à-dire les différents kerThemes. Les kerThemes étant à la fois composés de diagrammes de classe exécutables et de scénarios,

nous devons être capables de composer ces deux types de modèles. De plus, la composition entre deux *kerThemes* de base n'étant pas la même qu'entre un *kerTheme* de base et un *kerTheme* d'aspect, nous devons au total proposer quatre types de composition.

4.1. Composition de deux *kerThemes* de base

4.1.1. Composition des diagrammes de classe exécutable

La composition de deux diagrammes de classe exécutable de *kerThemes* de base, est essentiellement fondée sur la notion de “merge” de deux diagrammes de classes exécutables présentée dans le papier (Jackson *et al.*, 2006). Dans cette approche, afin de pouvoir paramétrer la sémantique de la composition de deux diagrammes de classe, Jackson *et al.* propose un *framework* de composition fixant le cœur de la sémantique du *merge* quand les deux diagrammes de classe exécutable ne possèdent aucun conflit. Puis pour chaque conflit potentiel, comme la présence de deux méthodes de même nom dans les deux diagrammes, l'utilisateur peut venir enregistrer un “*fixeur*” capable de résoudre ce conflit. Les choix retenus pour la résolution de conflits vont permettre de paramétrer la sémantique de la composition. Cette approche se rapproche des travaux sur les directives de composition (Straw *et al.*, 2004) où chaque “*fixeur*” définit une directive de composition. Ce *framework* de composition est plus bas niveau que les directives de composition mais il permet de travailler sur la structure mais aussi sur la partie comportementale du diagramme de classe exécutable.

4.1.2. Composition des scénarios

La composition de scénarios de deux *kerThemes* de base est en partie fondée sur la composition présentée dans Theme/UML (Clarke *et al.*, 2005) qui consiste principalement à insérer un comportement après ou avant une méthode spécifiée explicitement. Ce type de composition peut être exprimé avec l'opérateur classique de composition séquentielle faible (une définition peut être trouvée dans (Klein, 2006)). Les scénarios représentés sur les figures 1 et 2 décrivent deux scénarios de base. Leur composition est représenté par le scénario de la figure 6 (ce scénario contient également le résultat du tissage de l'aspect comportemental décrit plus loin). Les scénarios des figures 1 et 2 représentent deux comportements différents de la méthode *enroll*. Lorsque l'on compose ces deux scénarios, nous voulons conserver les deux comportements. Pour cela, on “colle” simplement (c'est la composition séquentielle faible) le scénario de la figure 2 après la méthode *display* (qui n'est pas dans une boucle) de la figure 1.

4.2. Composition d'un *kerTheme* d'aspect avec un *kerTheme* de base

4.2.1. Composition des diagrammes de classe exécutable

La composition d'un *kerTheme* d'aspect avec un *kerTheme* de base est quelque peu plus complexe que la composition de deux *kerThemes* de base. En effet cet opérateur de composition est lui aussi construit à l'aide du *framework* de *merge* paramétrable. Cependant, au moment de la définition du *kerTheme* d'aspect, le concepteur ne connaît pas le(s)

point(s) de jonction sur le(s)quel(s) il s’appliquera. Pour ce faire, il impose d’écrire le kerTheme d’aspect suivant certaines contraintes syntaxiques. Ainsi, une classe du kerTheme d’aspect doit hériter d’une classe abstraite Aspect comme illustrée dans la Figure 5.(a). Cette classe appelée “classe Aspect” dans le reste de ce papier est le futur point d’accroche avec le kerTheme de base. Le concepteur peut surcharger la méthode *pre* ou *post* pour ajouter du comportement avant ou après le point de jonction. Il peut aussi contrôler l’appel à la méthode de base à l’aide de l’appel à la méthode *proceed* de la “classe Aspect”. Dans l’exemple sur la persistance, un appel à la base de donnée est effectuée après l’appel au point de jonction comme illustrée figure 5.(b).

```

abstract class Aspect{
  operation _pre() : Void is do
    self.proceed
  end

  operation proceed() : Void is do
    self._post
  end

  end

  operation _post() : Void is do
  end
}

class Database {
  reference myAspect : MyAspect

  operation executeQuery(String sqlRequest) : Void
  is do
    myAspect.acknowledgement
  end

  operation commit() : Void is do
  end
}

class Persistency inherits Aspect{
  reference database : Database

  operation createRequest() : String is do
    //Code for reading mapping config files
    //and create the sql request
    ...
  end

  method _post() : Void is do
    database.executeQuery(self.createRequest)
  end

  operation acknowledgement() : Void is do
    database.commit
  end
}

```

(a) Exemple de kerTheme avant tissage

```

//Register after weaving
class Register {
  //Properties from the aspect
  reference database : Database
  //Properties from the join point class
  reference guest : Guest#register
  attribute register : Set<Object>

  //Methods from the aspect
  operation add() : Void is do
    self._pre_MyAspect_add
  end

  operation _pre_Persistency_add() : Void is do
    self._add
  end

  operation _add() : Void is do
    self._add1()
    self._add2()
  end

  end

  operation _add2() : Void is do
    self._post_Persistency
  end

  end

  operation _post_Persistency() : Void is do
    database.executeQuery(self.createRequest)
  end

  operation createRequest() : String is do
    //Code for reading mapping config files
    //and create the sql request
    ...
  end

  end

  operation acknowledgement() : Void is do
    database.commit
  end

  end

  //Methods from the join point class
  operation check(o : Object) : Void is do
    var info : Boolean init quest.getInfo
    if register.exists(obj | obj == o) then
      quest.alreadyExists
    else quest.notAlreadyExists
    end
  end

  end

  operation _add1() : Void is do
  end
}

```

(b) Résultat après tissage

Figure 5. Exemple d’un kerTheme avant et après tissage

La composition d’un kerTheme d’aspect avec un kerTheme de base est effectuée en quatre étapes :

- 1) Premièrement, la “classe Aspect” dans le kerTheme d’aspect est aplatie. Par conséquent, les méthodes *pre*, *proceed* et *post* sont intégrées à la “classe Aspect” si elles ne sont pas surchargées par les concepteurs du kerTheme.
- 2) Ensuite, la définition de la coupe désigne les points de jonction au sein du kerTheme de base. La méthode *pre* de la “classe Aspect” est renommée en *_preX* où *X* représente le

nom de la méthode du masque de point de jonction. La méthode *proceed* est renommée en *_X*. Une méthode *X* est intégrée au sein de la “classe Aspect”. Cette méthode appelle la méthode *_preX*. Le point de jonction est renommé en *_X*.

3) La troisième étape représente la phase de composition à proprement dit. Cette étape utilise le *framework de merge*. Suite aux étapes de transformation une et deux, un conflit entre la méthode *_X* de la classe Aspect du *kerTheme* d’aspect et la méthode *_X* du *kerTheme* de base est créé. Un “fixeur” renomme les deux méthodes *_X*, celle du point de jonction et celle de la “classe Aspect”. Ce “fixeur” crée une nouvelle méthode *_X* qui appelle premièrement la méthode *_X* de la “classe Aspect” puis la méthode *_X* du *kerTheme* de base. D’autres “fixeurs” peuvent alors mettre à jour le modèle si d’autres méthodes ou d’autres classes sont en conflits.

4) Finalement, l’opérateur de composition *merge* unifie le *kerTheme* de base et le *kerTheme* d’aspect.

4.2.2. Composition des scénarios

La composition de scénarios d’un *kerTheme* de base et d’un *kerTheme* d’aspect est une application directe du tissage proposé dans (Klein *et al.*, n.d.), puisque le scénario d’un *kerTheme* d’aspect est un aspect comportemental. Dans (Klein *et al.*, n.d.), un aspect est appelé aspect comportemental car il est spécifié avec un langage de modélisation dynamique. Un aspect est défini comme une paire de scénarios, un scénario pour l’expression de coupe, c’est-à-dire un scénario interprété comme un prédicat sur la sémantique des diagrammes de séquence satisfait par tous les points de jonction (spécification du comportement à détecter), et d’un second scénario représentant le comportement désiré aux niveaux des endroits préalablement détectés. Tout comme dans Aspect-J, où un aspect peut être inséré “avant”, “après” ou “autour” un point de jonction, avec l’approche définie dans (Klein *et al.*, n.d.), un *advice* peut indifféremment compléter un comportement détecté, le remplacer par un nouveau comportement, ou le supprimer complètement. La figure 6 représente le *kerTheme* regroupant les trois *kerThemes* *interface-utilisateur*, *inscription* et *persistance*. On peut remarquer que l’aspect comportemental a été tissé à deux endroits sur la figure 6 (aux deux “extrémités” du scénario).

4.3. Mise en œuvre des opérateurs de composition en Kermeta.

Kermeta est un langage de méta-modélisation développé par l’équipe Triskell à l’IRISA. Il est construit comme une extension d’EMOF 2.0² en y ajoutant un langage d’action permettant de spécifier la sémantique opérationnelle des méta-modèles. Le langage d’action est impératif et fondé sur le paradigme de la programmation par objets. Kermeta est statiquement typé et supporte l’héritage multiple. Finalement, du fait de sa simplicité, sa compatibilité avec EMOF et certaines constructions pratiques comme les clôtures lexicale, Kermeta est aussi approprié pour effectuer des actions sur des modèles.

Dans le cadre de ces travaux, le choix de Kermeta se trouve particulièrement approprié pour plusieurs raisons. Premièrement, un *kerTheme* étant construit comme un scénario as-

2. essential MOF

socié à un diagramme de classe exécutable, Kermeta permet de construire le méta-modèle des scénarios et peut être utilisé comme langage de description de diagrammes de classe exécutable. De plus, Kermeta de part sa simplicité, sa compatibilité avec EMF et les outils de méta-modélisation autour d'eclipse et certaines construction pratique comme les clôtures lexicale, Kermeta est approprié pour mettre en œuvre les opérateurs de composition décrits ci-dessus. En conclusion, le choix de Kermeta permet de fournir un environnement homogène aux concepteurs de kerThemes. En effet, la modélisation d'un diagramme de classe exécutable pour un kerTheme où la paramétrisation de la composition de kerTheme s'effectue à l'aide du langage Kermeta.

5. Tester des kerThemes

5.1. Processus pour le test

La modélisation par aspects s'appuyant sur les kerThemes décrits dans les sections précédentes permet de tester les différentes préoccupations modélisées. Le processus de test s'appuie sur les deux vues fournies sur le comportement de chaque kerTheme. Le diagramme de classe peut être exécuté et il est donc possible de simuler le comportement avec des données de test particulières. Par exemple, il est possible d'exécuter un cas de test qui appelle la méthode *submit* de la classe *Controller* du kerTheme *Interface utilisateur*. Le diagramme de séquences du kerTheme permet de construire un oracle pour le cas de test. En effet, cette vue offre une description complète des séquences de messages possibles au cours de l'exécution du kerTheme. Lorsqu'un cas de test est exécuté sur le diagramme de classe, il est possible de produire une trace correspondant à l'ensemble des messages échangés entre objets. L'oracle consiste alors à vérifier que la trace produite par le cas de test est incluse dans le diagramme de séquence du kerTheme. L'inclusion de trace peut se faire grâce aux algorithmes présentés dans (Muscholl, 1999) (ou en utilisant les algorithmes de détection présentés dans (Klein, 2006)).

Le processus de test pour un kerTheme suit les étapes suivantes :

- instrumenter le diagramme de classe exécutables pour obtenir une trace
- exécuter le diagramme de classe avec une donnée de test et récupérer la trace
- comparer la trace obtenue avec l'ensemble des traces licites modélisées par le diagramme de séquence
- si la comparaison de trace révèle une incohérence entre la trace obtenue et la trace attendue, alors une erreur a été détectée et il faut la localiser

La difficulté majeure pour tester chacun des modèles de préoccupations tient au fait que certains modèles ne peuvent pas être validés sans être composés avec d'autres modèles au préalable. C'est évident pour les kerThemes d'aspect qui ne décrivent que des canevas de comportement et ne peuvent donc pas être exécutés sans être composés avec un autre modèle. Cela peut aussi se produire pour un kerTheme de base qui nécessite l'exécution d'un ou plusieurs kerThemes avant de pouvoir s'exécuter. Le processus de test doit prendre en compte cette difficulté pour l'exécution des tests et la localisation des erreurs.

Certains modèles de `kerThemes` de base peuvent être testés isolément. Ceci permet d’être sûr que si une erreur est détectée au cours du test, elle provient de ce `kerTheme`. Les erreurs ainsi détectées peuvent être corrigées dans ce modèle avant de le composer avec d’autres modèles. Par exemple, le `kerTheme` *Interface utilisateur* peut être testé isolément. Lorsqu’un modèle ne peut pas être testé isolément il faut tester le résultat de la composition de ce `kerTheme` avec d’autres `kerThemes`. La difficulté dans ce cas est de détecter la source de l’erreur. Elle peut être dans un des `kerThemes` qui ont été composés ou dans la spécification de la composition. Pour faciliter la localisation de l’erreur, nous pouvons profiter de la composition strictement incrémentale des `kerThemes`. En effet, les `kerThemes` ne sont composés qu’un par un. Ainsi lorsqu’un `kerTheme` est composé à un autre modèle, celui-ci a été testé dans une étape précédente. Dans ce cas, l’erreur détectée dans le modèle composite doit provenir soit du `kerTheme` qui vient d’être composé, soit de la spécification de la composition de ce `kerTheme` avec le modèle.

5.2. Détection d’erreurs dans les modèles

Cette approche de test nous permet de détecter plusieurs types d’erreur. Dans un `kerTheme` de base ou un `kerTheme` d’aspect, le test permet de mettre en évidence des appels de méthode manquant ou des séquences d’appels illicites. Le test sur un modèle composé permet de détecter des erreurs introduites lors de la spécification de la composition de deux `kerThemes`. Lorsque deux `kerThemes` de base sont composés, le test du modèle composé permet de détecter des erreurs dans la spécification de l’ordre dans lequel les `kerThemes` doivent être composés. Lorsque deux `kerThemes` d’aspect sont composés, le test permet de détecter une erreur dans l’expression du point de jonction.

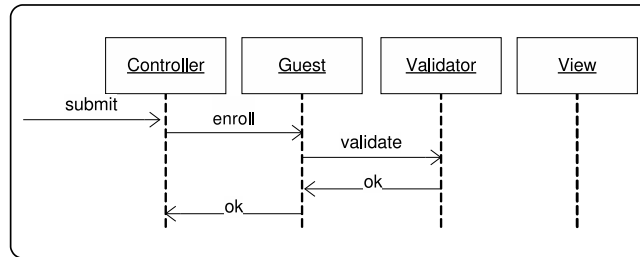
```
class Guest{
  operation enroll()
  operation ok() : Void is do
    controller.ok
    view.display(void)
  end
  operation nok()
```

Figure 7. Une erreur dans le `kerTheme UI`

Pour illustrer l’approche de détection d’erreur dans un `kerTheme` de base isolé, nous utilisons l’exemple de la figure 7. Cette figure reprend le code pour le `kerTheme` *Interface utilisateur* qui est décrit figure 4 dans lequel une erreur a été introduite : l’appel de la méthode `display` vers l’objet `view` a été supprimé. L’exécution de ce `kerTheme` *Interface utilisateur* erroné produit la trace d’exécution de la figure 8.(a). Comme cette trace est une séquence simple d’appels de méthode, elle peut être facilement traduite en un diagramme de séquence tel qu’illustré figure 8.(b). Graphiquement, il apparaît que ce diagramme de séquence n’est pas une trace licite, par comparaison au diagramme de séquence attendu (figure 1). Cette incohérence entre le comportement attendu et le comportement observé révèle une erreur. La comparaison entre deux diagrammes de séquence peut être automatisée grâce aux algorithmes décrits dans (Muscholl, 1999).

Controller.submit
 Guest.enroll
 Validator.validate
 Guest.ok
 Controller.ok
 Controller.ok\$
 Guest.ok\$
 Validator.validate\$
 Guest.enroll\$
 Controller.submit\$

(a)



(b)

Figure 8. Une trace d'exécution erronée pour le kerTheme UI

6. Travaux connexes

Le but principal du travail décrit dans ce papier étant la validation de modèles de préoccupations, cette section présente quelques travaux liés à la validation de modèles. Nous présentons également, mais plus brièvement, quelques travaux sur la validation de programmes par aspects.

Dans (Gogolla *et al.*, 2005) les auteurs s'intéressent à la validation de diagrammes de classe UML comportant des contraintes OCL. Les auteurs proposent l'outil USE qui permet d'interpréter des contraintes OCL et d'assurer la conformité d'un diagramme d'objets à un diagramme de classe comportant des contraintes OCL. Pour valider un modèle statique UML, le testeur doit construire et animer des diagrammes d'objets et l'outil permet de vérifier qu'aucune contrainte n'est violée. Afin de valider leur approche, les auteurs l'ont appliquée à une partie du méta-modèle UML et aux contraintes de bonne formation qui lui sont associées. Dans la technique initialement proposée, les diagrammes d'objets sont construits manuellement par le testeur. Dans (Gogolla *et al.*, 2005) les auteurs s'intéressent à automatiser la construction de diagrammes d'objets à partir d'une description déclarative des structures attendues.

Dans (Andrews *et al.*, 2003, Ghosh *et al.*, 2003), Andrews et al. s'intéressent au test de modèles de conception UML. Les modèles UML sont rendus exécutables grâce à un langage d'action proche des diagrammes d'activités. Les auteurs définissent des critères de test qui s'appuient sur les diagrammes de classe et les diagrammes de collaborations du modèles UML. L'idée des critères proposés est d'assurer la couverture des structures des diagrammes de classe et des collaborations. Les scénarii de test sont générés sous la forme de diagrammes de séquences qui peuvent être complétés par des assertions. Une technique de génération de tests permettant de satisfaire ces critères a été implantée dans un outil présenté dans (Dinh-Trong *et al.*, 2005).

Peu de travaux s'intéressent au test dans le cadre du développement par aspects. Quelques papiers traitent ce problème au niveau programmation. Dans (Xu *et al.*, 2006), les auteurs proposent de générer des tests pour un programme par aspects à partir d'automates modélisant le comportement du programme de base et des aspects. Ils s'appuient

sur une composition incrémentale pour détecter les fautes introduites par un aspect et les fautes présentes dans le programme de base. Xie et al. (Xie *et al.*, 2006) s'intéressent à tester le comportement d'un aspect décrit avec AspectJ. Ce travail consiste à développer un environnement qui permet de réutiliser des outils de génération de test pour Java avec des aspects. Enfin, (Alexander *et al.*, 2004) proposent des modèles de fautes pour les programmes AspectJ qui permettent de valider des cas de test dédiés aux aspects.

7. Conclusion

Ce document présente une étude autour de l'introduction du test dans la modélisation par aspects. La principale contribution de ces travaux est de permettre la détection, dès les phases amont, des erreurs dans les modèles des préoccupations. Le test peut se résumer en une vérification de cohérence entre deux vues comportementales d'un même système. Au travers de cette expérience, les conclusions sont diverses. Tout d'abord, si la prise en compte des différentes préoccupations du système et leur décomposition dès les phases amont du développement logiciel est un problème clairement identifié par la communauté. Peu d'outils ou de canevas de modélisation propose réellement des opérateurs de composition de modèles pour faciliter le tissage des différents modèles représentant les préoccupations d'un système. Ce manque nous a poussé dans kerTheme a travaillé sur ces opérateurs de compositions afin de bénéficier d'un socle suffisant afin de pouvoir mener des expérimentations. Par conséquent, plusieurs opérateurs de composition et de tissage ont été développés pour fournir un environnement de modélisation par aspects adapté fournissant les caractéristiques requises pour le test.

La complémentarité entre un diagramme de séquence et un diagramme de classe exécutable permet de décrire finalement le même système mais à l'aide d'un point de vue très différent. L'expression de la composition et les expressions de coupe par exemple ne possède pas exactement le même pouvoir d'expression. Ainsi, une expression de coupe définie au niveau du diagramme de séquence sera dans une certaine mesure plus expressive et plus simple à définir. Cette complémentarité permet aussi de vérifier que la composition si elle est exprimée par des moyens différents représente au final le même système.

Les travaux futurs s'intéressent à deux pistes. Premièrement quantifier l'apport de l'introduction du test dans la modélisation par aspects, l'objectif est alors de mesurer le nombre d'erreur identifié dès les phases amont et la simplicité de les résoudre dès les phases amont plutôt que dans les phases de développement ou d'intégration. Deuxièmement, il reste à travailler sur la traçabilité dans le cadre de composition mettant en œuvre plus de deux kerThemes. En effet, afin que l'identification d'une erreur soit pertinente pour un concepteur, il est, en outre, nécessaire de pouvoir localiser précisément l'origine de cette erreur qui peut être due à une erreur dans une des préoccupations, dans l'expression de la composition ou du à une incompatibilité entre des préoccupations du système telle qu'elles ont été pensées en isolation.

8. Bibliographie

- Alexander R. T., Bieman J., Andrews A., Towards the systematic testing of aspect-oriented programs, PhD thesis, Colorado State University, 2004.
- Andrews A., France R., Ghosh S., Craig G., « Test adequacy criteria for UML design models », *Software Testing, Verification and Reliability*, vol. 13, n° 2, p. 95 -127, 2003.
- Clarke S., Baniassad E., *Aspect-Oriented Analysis and Design : The Theme Approach*, number ISBN : 0-321-24674-8, Addison Wesley, 2005.
- Dinh-Trong T., Kawane N., Ghosh S., France R., Andrews A., « A Tool-Supported Approach to Testing UML Design Models », *ICECCS'05*, Shanghai, China, 2005.
- Fleurey F., Langage et méthode pour une ingénierie des modèles fiable, PhD thesis, Université Rennes 1, 2006.
- Ghosh S., France R., Braganza C., Kawane N., Andrews A., Pilskalns O., « Test Adequacy Assessment for UML Design Model Testing », *ISSRE'03 (Int. Symposium on Software Reliability Engineering)*, Denver, CA,USA, p. 332-343, 2003.
- Gogolla M., Bohling J., Richters M., « Validating UML and OCL Models in USE by Automatic Snapshot Generation », *Software and Systems Modeling*, vol. 4, n° 4, p. 386-398, 2005.
- IEEE, IEEE Standard Glossary of Software Engineering Terminology, Technical report, IEEE, New York, September 2006.
- ITU, *Recommendation Z.120 : Message Sequence Chart (MSC)*, E Rudolph (ed.), Geneva, 1996.
- Jackson A., Barais O., Jezequel J.-M., Clarke S., « Towards a Generic and Extensible Merge Operator », *Second Workshop on Models and Aspects, Handling Crosscutting Concerns in MDS at ECOOP 06*, Nantes, France, July, 2006.
- Klein J., Aspects Comportementaux et Tissage, PhD thesis, Université Rennes 1, 2006.
- Klein J., Fleurey F., « Tissage d'Aspects Comportementaux », *Langages et Modèles à Objets : LMO'06*, Nimes, France, 2006.
- Klein J., Hérouët L., Jézéquel J., « Semantic-based Weaving of Scenarios », in , R. E. Filman (ed.), *Proceedings of the 5th International Conference on Aspect-Oriented Software Development, AOSD 2006, Bonn, Germany, March 20-24, 2006*, n.d.
- Lewis G., Wraga L., Approaches to Constructive Interoperability (CMU/SEI-2004-TR-020 ESC-TR-2004-020), Technical report, Pittsburgh, PA : Software Engineering Institute, Carnegie Mellon University, 2005.
- Mukerji J., Miller J., Technical Guide to Model Driven Architecture : The MDA Guide v1.0.1., Technical report, OMG's Architecture Board, 2003.
- Muller P.-A., Fleurey F., Jézéquel J.-M., « Weaving Executability into Object-Oriented Meta-Languages », *Proc. of MODELS/UML*, LNCS, Jamaica, 2005.
- Muscholl A., « Matching Specifications for Message Sequence Charts », *Proceedings of FOS-SACS'99*, LNCS 1578, p. 273-287, 1999.
- Straw G., Georg G., Song E., Ghosh S., France R., Bieman J. M., « Model Composition Directives », in , T. Baar, , A. Strohmeier, , A. Moreira, , S. J. Mellor (eds), *UML 2004 - The Unified Modeling Language. Model Languages and Applications. 7th International Conference, Lisbon, Portugal, October 11-15, 2004, Proceedings*, vol. 3273 of LNCS, Springer, p. 84-97, 2004.
- Xie T., Zhao J., « A Framework and Tool Supports for Generating Test Inputs of AspectJ Programs », *AOSD'06 (International Conference on Aspect-Oriented Software Development)*, Bonn, Germany, p. 190-201, 2006.

Xu D., Xu W., « State-Based Incremental Testing of Aspect-Oriented Programs », *AOSD'06 (Aspect Oriented Software Development)*, Bonn, Germany, 2006.