Réparation de modèles par transformations indempotentes

Clémentine Nemo, Mireille Blay-Fornarino, Johan Montagnat, Michel Riveill

Réparation de modèles par transformations idempotentes

Clémentine Nemo, Mireille Blay–Fornarino, Johan Montagnat, Michel Riveill

Université de Nice – Sophia Antipolis
Laboratoire I3S (CNRS - UNSA), Équipe Modalis
Bâtiment Polytech'Sophia – 930 route des Colles
B.P. 145, F-06903 Sophia Antipolis Cedex
{blay,nemo,johan,riveill}@polytech.unice.fr

RÉSUMÉ. *Dans le cadre d'un développement dirigé par les modèles, les applications logicielles se définissent par utilisation de transformations. Certaines transformations visent à répondre à des "problèmes récurrents" et leurs applications garantissent le respect de contraintes dans le modèle résultant. Elles sont identifiées sous le terme de* ***Politiques****. L'intérêt d'une conception par applications de politiques est à la fois de faciliter la tâche du concepteur et de garantir par construction la cohérence du modèle construit. Nous étudions dans cette article la relation qui unit politiques et évolution de modèles en nous penchant sur la cohérence du modèle construit. Nous proposons pour cela une démarche de modélisation des politiques qui vise à assurer la cohérence du modèle construit relativement à chaque politique appliquée et à réparer automatiquement ces modèles dans le cas contraire. La réparation est alors basée sur la réapplications des politiques.*

ABSTRACT. *Several model transformations automatically addressed some "recurrent problems". They guarantee the contract verification of the resulting model. We name these transformations* ***Policies****. Using policies enables a quick construction of the models and guarantees the conformity to the resulting model. In this article we study the overlap between the policies introduced and its impact on the model evolution regarding its conformity. We defend an approach to guarantee and restablish the conformity of the resulting model by re-introduction of policies.*

MOTS-CLÉS : *Évolution, Politique, Transformation, Réparation automatique, Conformité des modèles*

KEYWORDS: *Evolution, Policy, Transformation, Automatic Repair, Model conformity*

[e] soumission à *RSTI*, le 27 mai 2011.

1. Introduction

La place prépondérante occupée par les systèmes d'information (SI) dans les entreprises font de la maintenance et de l'évolution de ceux-ci un enjeu majeur du développement logiciel. Dans sa capacité à supporter et à exécuter les processus de l'entreprise, le SI se construit autour d'un modèle métier qui en structure les échanges. Le passage au système informatisé est alors confronté aux problèmes classiques des infrastructures distribuées pour prendre en charge des propriétés telles que la sécurité des accès aux données ou la tolérance aux pannes, que nous désignerons sous le vocable de *"politiques"* de sécurité, de réplication, ... Or dans le même temps la flexibilité est une des qualités plébiscitées des SI et consiste à pouvoir modifier ou étendre les fonctions rendues par le système sans évidemment perdre en qualité de service [CAS 08]. Accompagner l'évolution du système d'information en garantissant que l'architecture qui le supporte respecte bien l'ensemble des politiques et automatiser la réparation éventuelle, tels sont les objectifs visés par cet article.

L'automatisation dans le processus de développement semble dans ce contexte un moyen efficace et sûr pour permettre l'introduction de propriétés non fonctionnelles dans une architecture métier du SI. En effet, le développement dirigé par les modèles cible une production itérative de l'application par enrichissements successifs constitués par des modifications des modèles et des transformations automatiques. Ainsi, dans la mouvance des travaux autour de la séparation des préoccupations en programmation, différentes approches défendent aujourd'hui l'enrichissement des modèles par l'utilisation de patrons et transformations [MUL 05, BAR 08, CLA 01]. Ces approches reposent alors sur un processus séquentiel de construction dans lequel une fois les transformations appliquées (i) soit le modèle résultat n'est plus modifié et consitutue donc le résultat attendu, (ii) soit il n'est pas possible de garantir que les modifications ultérieures ne brisent pas la logique associée aux patrons. La première solution s'oppose à un développement par incréments [BAT 04] où les entités ajoutées doivent à leur tour être enrichies par exemple pour sécuriser un système de trace. La seconde solution rejoint les problèmes de gestion de la cohérence dans la construction collaborative de modèles en forçant a priori le respect d'un ensemble de contraintes pré-établies [BLA 09]. **Aucun de ces travaux n'aborde simultanément l'évolution par ajout de nouvelles politiques (et donc une modification itérative des contraintes à vérifier) et la réparation des modèles en cas de détection d'incohérences.** Or la complexité des politiques et de leurs impacts sur le modèle métier, de même que leur nature étrangère au métier pour celui qui introduit la politique rend la correction des modèles d'autant plus difficile qu'il n'en maîtrise qu'une représentation partielle [GIR 06].

Nous proposons de prendre en charge l'évolution de la modélisation par des actions utilisateurs et des transformations qui forcent le respect de contraintes portées par ces politiques, *par exemple ce composant n'accepte que des requêtes authentifiées*. Pour assurer le respect de ces contraintes (*i.e.* la cohérence du modèle), l'application d'une politique modifie le modèle par l'introduction de nouveaux éléments, *par exemple une politique d'authentification liée à un composant ajoute une autorité de certification pour gérer les certificats et force la présence d'un paramètre certificat associé à toutes les opérations exposées par le composant via ses interfaces.* Si l'application d'une politique sur un modèle assure le respect des contraintes qu'elle définit, il est bien entendu que l'application d'un ensemble de politiques induit le respect de toutes les contraintes portées par les politiques. Notre objectif est donc d'assurer la cohérence du modèle résultant de l'application de plusieurs politiques afin qu'il respecte l'ensemble des contraintes associées, et le cas échéant, de permettre sa réparation par modification automatique des éléments du modèle. Nous défendons alors que le travail réalisé dans l'écriture de la transformation qui définit une politique est le même que celui réalisé pour réparer le modèle, à la différence près que seules les actions utiles doivent être évaluées. Nous appelons ces transformations, *des transformations idempotentes*, dans la mesure où lorsqu'elles sont appliquées sur un modèle qui respecte les contraintes à satisfaire elles n'ont aucune action.

L'apport de cet article est alors (i) de formaliser la notion de transformation idempotente et de contexte d'application afin de permettre la réparation des modèles, (ii) de proposer un système de transformations assurant par construction l'idempotence et (iii) de définir un processus de réparation par ré-application des transformations.

Dans un premier temps nous précisons le contexte de cette étude au travers d'un rapide état de l'art et posons le problème abordé au travers d'un exemple fil rouge (cf. §2). Nous explicitons ensuite la notion de modèle et de conformité (cf. §3) avant de formaliser les transformations idempotentes (cf. §4). Le processus d'évaluation des transformations idempotentes est alors décrit en montrant en particulier comment

nous utilisons la notion de contexte pour supporter la réparation des modèles (cf. §5). Sur cette base, nous présentons une stratégie séquentielle puis ensembliste d'évaluations des transformations (cf. §6). La mise en oeuvre de la proposition est alors brièvement exposée ainsi que les évaluations qui ont été réalisées. Les perspectives à ce travail sont ensuites décrites (cf. §7). Enfin la section 8 conclut cet article.

Ce travail a été conduit dans le cadre d'une collaboration avec DCNS[1] .

2. Politiques : État de l'art et exemple

2.1. *Positionnement*

Une politique est définie comme un plan d'actions permettant de préserver ou d'associer certaines capacités à un assemblage [BEI 07].[2] La prise en compte des politiques intervient à différents stades du développement logiciel de la phase de collecte des exigences au déploiement du code. L'expression et l'utilisation des politiques prennent ainsi différentes formes.

En phase de production de code, l'utilisation des frameworks cible la vérification des contraintes liées à une politique dans le code. Par exemple l'utilisation des frameworks *log4j* ou *JAAS* permet de satisfaire des politiques relatives à la gestion de la trace et de l'authentification des utilisateurs par l'intégration au niveau du code de classes dédiées. Dans le domaine des composants et des services [BEI 07, W3C 07], des fichiers de configuration et des annotations sont alors une manière de spécifier l'association des politiques aux composants et de garantir le respect des contraintes liées. La compatibilité des différentes politiques est établie au niveau des spécifications. L'ajout d'une nouvelle politique implique de déterminer les incompatibilités potentielles avec les autres politiques supportées par le framework. L'utilisation conjointe de plusieurs frameworks reste un problème difficile à gérer. Pour faciliter l'introduction des politiques au niveau du code et gérer certaines compositions, la programmation par aspects procède par une expression séparée des politiques à intégrer et un processus de tissage des codes [CHA 04, KIC 97]. On attend de l'application des aspects qu'elle assure le respect des contrats pour lesquels ils ont été définis [PAW 05].

Cependant, les préoccupations non fonctionnelles apparaissent bien plus tôt dans le développement du logiciel et plusieurs travaux s'intéressent aujourd'hui à leur introduction dans la phase de gestion des exigences et la modélisation de l'architecture [BAN 06]. Ainsi en phase de modélisation, nous trouvons différents profiles, templates ou patrons qui permettent de modifier les modélisations architecturales afin de cibler l'application de politiques via des ateliers [GUE 00]. A partir d'annotations sur un modèle de départ, de directives de composition ou d'aspects, une architecture plus élaborée est automatiquement générée [CAR 03, SIM 05, BAR 06, LOD 02]. Garantir la cohérence des modèles alors générés et plus précisément supporter l'évolution de ces modèles en assurant la préservation de la cohérence est un des défis qui reste à relever.

Parmi les pistes pour répondre à ce problème, l'analyse des transformations elles-mêmes est une voie prometteuse. Dans ce cas le modèle évolue par application de transformations qui ciblent le respect d'un ensemble de contraintes, rassemblées dans un métamodèle cible [ENG 02] ; le domaine du refactoring est particulièrement pourvoyeur de telles transformations [MEN 02, VAN 03]. Des stratégies d'application des transformations permettent alors de contrôler et de composer l'application des transformations elles-même [WAG 08]. En considérant les politiques comme des transformations, nous cherchons à construire un modèle qui respecte les contraintes portées par chaque politique et donc la conformité du modèle relativement à l'ensemble des métamodèles ciblés. Atteindre ce but nécessite alors de (B1) vérifier la cohérence d'un modèle relativement à chaque métamodèle ciblé et de (B2) savoir réparer le modèle en cas de conformité non vérifiée. Le premier point (B1) est abordé dans la littérature en vérifiant la conformité du modèle résultant relativement au résultat de la composition des métamodèles ciblés [STR 04, CLA 02]. Dans notre approche chaque politique met en place des éléments propres à la vérification de ses contraintes, relatifs par exemple à des fichiers de journalisation ou une autorité de certification. Ces éléments correspondent à des métaéléments spécifiques. Afin de garder la possibilité d'introduire de nouvelles politiques (venant

1. http ://en.dcnsgroup.com/
2. "The term Policy is used to describe some capability or constraint that can be applied to service components or to the interactions between service components represented by services and references. An example of a policy is that messages exchanged between a service client and a service provider be encrypted, so that the exchange is confidential and cannot be read by someone who intercepts the conversation."

de l'utilisation d'un nouveau framework ou de spécifications logicielles supplémentaires) nous choisissons de définir la cohérence d'un modèle par la conformité relativement à chaque métamodèle cible. Nous ne cherchons donc pas la construction d'un métamodèle cible unique. Le second point (B2) est lié aux diagnostiques fait lors de la validation. L'expression des contraintes des politiques sous forme de contrats [OMG 10] permet l'utilisation de vérificateurs [CRÉ 10] ou de transformations [BÉZ 05]. Le retour de la validation est alors un indicateur pour guider l'utilisateur dans la réparation. L'utilisation de stratégies de réparation prédéfinies [SIL 10] permet d'automatiser cette réparation. La réparation d'un modèle revient alors à exécuter une suite d'actions sur les éléments du modèle afin que le résultat soit conforme au métamodèle ciblé. Si l'on considère l'écriture d'une transformation aussi comme une liste d'actions permettant d'obtenir un modèle conforme au métamodèle ciblé [BAU 06], nous défendons le fait que chaque transformation décrit déjà les actions de réparation. Sur ce constat, et afin de ne pas écrire les contraintes permettant de vérifier la conformité, nous cherchons alors à spécifier l'application d'une transformation qui lors de ses applications n'exécute que les actions nécessaires. Nous les appelons **transformations idempotentes**. Dans le contexte d'une approche collaborative garantir le modèle résultant revient alors à ré-appliquer l'ensemble de transformations. Différents travaux permettent d'appliquer plusieurs fois la même transformation de manière idempotente en utilisant des modes spécifiques au langage de transformation [STE 10, LAJ 10, MEN 06]. L'idempotence est alors prise en charge par le développeur de la transformation, ce qui est une tâche difficile et qui suppose une expression exhaustive des différentes conditions d'application des transformations. Nous proposons de définir un système qui supporte de manière intrinsèque l'idempotence.

Nous présentons à présent notre problématique au travers d'un exemple qui nous servira à illustrer notre proposition.

2.2. *Un exemple fil rouge : Application sécurisée de partage d'images médicales*

Pour étayer notre étude nous utilisons un intergiciel de partage d'images médicales issu du projet ANR NeuroLOG (ANR-06-TLOG-024). qui vise à développer une plate-forme logicielle distribuée pour aider la communauté des chercheurs en neuro-imagerie à mutualiser des images et des programmes de traitement d'images [GAI 09]. Chaque site (centre de neurosciences) partenaire de la fédération NeuroLOG administre une base de données stockant des données (images médicales), des métadonnées (informations relatives aux images médicales) et des processus relatifs aux problématiques des neurosciences. L'ensemble des informations est partagé grâce à une gestion globale des sites. Le caractère privé des données ainsi distribuées exige alors un contrôle des accès à ces données. Dans cet exemple nous montrons l'évolution d'un modèle par actions de l'utilisateur et applications de politiques puis sa réparation par ré-application des politiques.

2.2.1. *Modélisation d'un site de l'application NeuroLOG*

La figure 1 est composée d'un diagramme à composants `UML`[3] extrait de l'application NeuroLOG qui ne prend pas en compte les préoccupations de sécurité. Afin de restreindre l'exemple nous avons choisi de modéliser un seul site et son système de stockage local des informations.

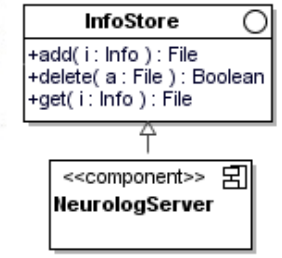


Figure 1. *Modélisation d'un site de l'application NeuroLOG (modèle initial)*

Le serveur de chaque site partenaire (`NeurologServer`) fournit par une interface (`InfoStore`) la possibilité aux utilisateurs d'ajouter, de télécharger et de supprimer des informations (`Info`). Sur la base de cette architecture initiale, nous proposons d'intégrer des préoccupations relatives à la sécurisation des accès aux données [CHA 08]. Ces préoccupations sont définies par des politiques. Notre but est de passer d'une

3. [NEM 10a] propose un exemple plus large

application de partage d'images médicales à **une application sécurisée de partage d'images médicales** [NEM 09].

2.2.2. *Politiques de sécurisation des accès aux données d'un site NeuroLOG*

Contrôle des accès : Le serveur Neurolog est le composant qui permet d'atteindre les données médicales. Afin de mieux gérer les accès au serveur, l'utilisation d'un objet de délégation (proxy) est requise. La mise en place d'un proxy [GAM 93] cible le respect des contraintes suivantes :

```
Le composant proxy fournit les mêmes interfaces que le composant auquel
il se substitue. L'utilisation du proxy implique une redirection des
références requérant les interfaces fournies du composant. Le composant
auquel il se substitue fournit une interface donnant des indications sur
le composant proxy.
```

Le modèle figure 2 représente un modèle de l'application NeuroLOG dans lequel le contrôle d'accès aux données est modélisé. Dans la suite de cet article, nous faisons référence à cette politique en utilisant la notation P_{proxy}.

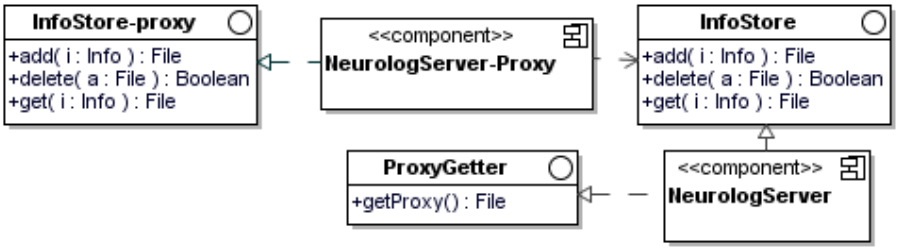


Figure 2. *Modélisation intégrant un contrôle d'accès aux données (P_{proxy})*

Journalisation des Accès : Dans le but de suivre l'activité d'un utilisateur malveillant en cas de problème de sécurité, un système de journalisation est mis en place sur chaque site de la fédération NeuroLOG [HA 02]. Les fichiers de journalisation contiennent entre autre les demandes d'authentification, d'accès aux données ou de soumission des requêtes. Les fichiers sont stockés dans une base de données prévue à cet effet. La mise en place de la journalisation des accès cible le respect des contraintes suivantes[4] :

```
La journalisation des accès aux opérations fournies par un composant
implique un accès par ce composant à une base de données dans laquelle
tous les accès sont tracés. Une interface fournie par le composant permet
l'accès aux fichiers de journalisation.
```

Le modèle figure 3 représente la modélisation d'un site de l'application NeuroLOG introduisant la journalisation des accès au serveur. Dans la suite de l'article, nous faisons référence à cette politique en utilisant la notation P_{trace}.

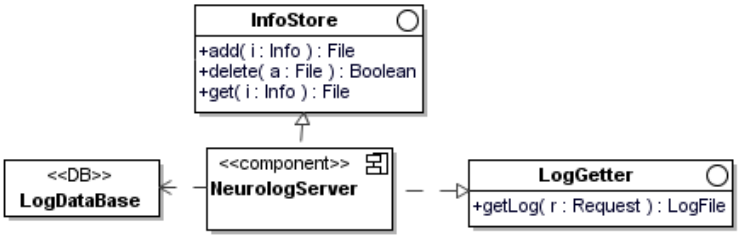


Figure 3. *Modélisation intégrant la journalisation des accès au serveur (P_{trace})*

4. Nous nous restreignons aux aspects structurels.

2.2.3. *Modèle visé*

A partir du modèle initial présenté figure 1 nous visons la modélisation présentée figure 4. Cette modélisation représente l'introduction des deux politiques présentées précédemment mais aussi de l'ajout d'éléments par l'utilisateur, que nous appelons *actions utilisateurs*. Ces éléments sont liés à l'ajout de deux fonctionnalités : la gestion (investigation, calcul de statistiques, ...) des fichiers de journalisation (ajout du composant `LogManager`) et la mise-à-jour des données médicales (ajout de l'opération `update`). Nous notons $A_{logManager}$ et A_{update} ces actions. Ce modèle final respecte les contraintes ciblées par les deux politiques. Notre objectif est d'obtenir ce résultat quel que soit l'ordre d'introduction des politiques et d'exécution des actions utilisateurs.

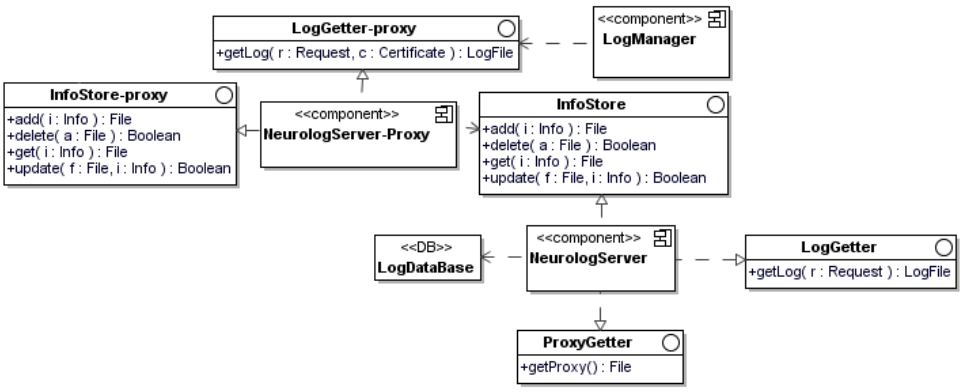


Figure 4. *Modélisation d'un site Neurolog sécurisé (P_{proxy}, P_{trace}) avec la gestion des traces ($A_{logManager}$) et la mise-à-jour des données (A_{update})*

2.2.4. *Introductions de politiques et réparation de modèle*

Nous nous basons sur cet exemple pour illustrer la réparation d'un modèle par ré-application des transformations. La figure 5 représente la construction d'un modèle par application de deux politiques P_{proxy} et P_{trace} et en détaillant chaque étape par un modèle intermédiaire. Le modèle *m0* représente le modèle initial. Le modèle *m1* représente le modèle construit à partir de *m0* et par introduction de la politique P_{proxy}. Le modèle *m2* représente le modèle construit par introduction de la politique P_{trace} à partir du modèle *m1*. On peut observer que le modèle *m2* ne respecte alors plus les contraintes liées à la politique P_{proxy} : le composant `NeurologServer-proxy` ne fournit pas les mêmes interfaces que le composant auquel il se substitue puisque l'interface `LogGetter` créée par l'action P_{trace} n'est pas prise en compte par la politique de contrôle des accès.

Dans le cas présenté il est évident qu'une modification de l'ordre d'application permet d'arriver au modèle souhaité. Cependant la modélisation d'un système comporte des phases d'itérations [GER 02] où un ensemble de politiques, issues de nouvelles exigences ou de raffinement des besoins, sont introduites au fur et à mesure de l'évolution du modèle. Il n'est alors pas réaliste de calculer l'ordre d'introduction de toutes les politiques, car à chaque étape de l'évolution cet ensemble peut être modifié.

Il est alors tentant de ré-appliquer la transformation liée à la politique dont les contraintes ne sont plus vérifiées. Cependant sans propriété sur l'application de la transformation, toutes les interfaces exposées par le composant `NeurologServer` seront dupliquées, i.e. les interfaces `LogGetter` et `InfoStore` comme dans le modèle *m3*. Nous avons alors besoin de qualifier l'application de la transformation par une propriété d'idempotence afin que l'application de la même transformation sur un modèle ne mette en place que les éléments nécessaires, sans dupliquer les éléments existants. Avec une telle sémantique sur l'application de la transformation, la réparation du modèle est automatique par ré-application. Le modèle *m4* représente le modèle résultant de la ré-application de la politique P_{proxy} en considérant l'application idempotente. Seule l'interface `LogGetter-proxy` est ajoutée. Le modèle *m4* est alors conforme à la politique P_{proxy}.

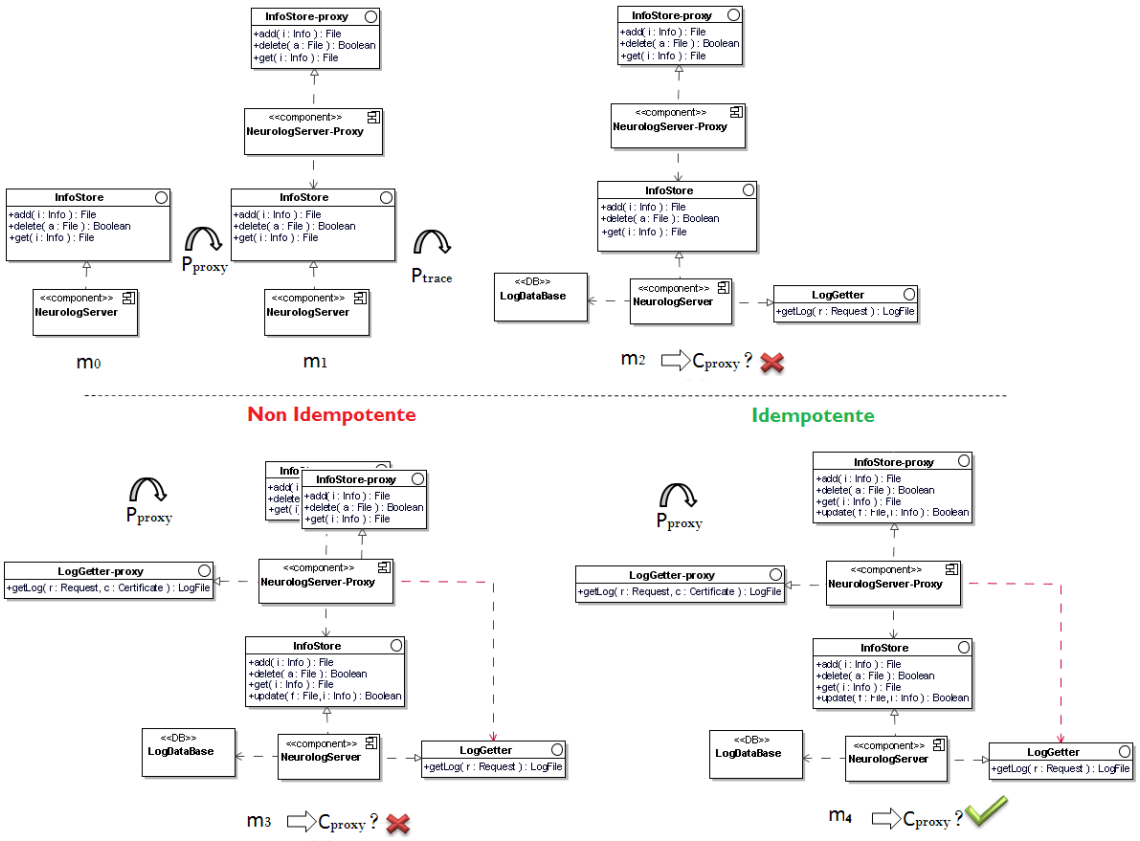


Figure 5. *(Dessus) modèle m2 résultat de l'application de P_{proxy} et P_{trace} et (dessous) modèle m4 souhaité*

3. Cohérence d'un modèle relativement à plusieurs métamodèles

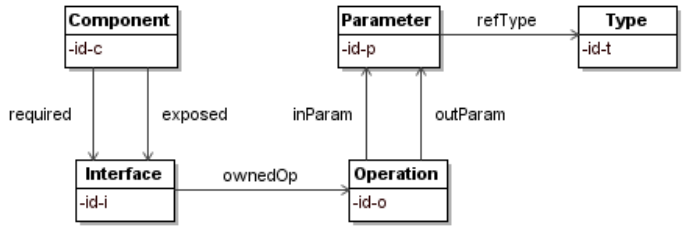
Nous attendons de l'application de politiques à un modèle que le modèle résultant respecte les contraintes portées par chacune des politiques. Afin d'expliciter cette notion, nous introduisons à présent la notion de cohérence d'un modèle. Nous proposons une modélisation minimale qui permet de supporter l'association de plusieurs métaéléments à un même élément (cf. §3.1). Nous basons alors la définition de la conformité d'un modèle relativement à plusieurs métamodèles sur cette modélisation (cf. §3.3). Cette définition sert de base à la définition de la cohérence d'un modèle construit par application de plusieurs politiques.

3.1. *Métamodèle : Définition*

Un métamodèle est composé d'un ensemble de métaéléments. A tout métaélément est associé un identifiant et un ensemble, éventuellement vide, précisant ses super-métaéléments. Nous notons MM l'ensemble des métamodèles, ME l'ensemble des métaéléments et ID l'ensemble des identifiants.

Définition 3.1 (Métamodèle) *Un métamodèle est un triplet $< mm, super, ident >$ avec :*
mm=$\{me_1, \ldots, me_n\}$ un ensemble de métaéléments tel que pour tout $me \in mm$,
super(me) = $\{me_{super} | me_{super} \in mm\}$ ou $\varnothing$,
ident(me) = i, $i \in ID$.

La figure 6 illustre la définition d'un métamodèle sous la forme d'un diagramme UML et la correspondance avec notre formalisation.



mm_{comp}={Component,Interface,Operation,Parameter,Type,exposed,required,ownedOp,inParam,outParam, refType,reference}.
super(exposed)= {reference}, super(required)={reference}, super(ownedOp)={reference}, super(inParam)={reference}, super(outParam)={reference}, super(refType)={reference}

Nous détaillons seulement les relations super *pour les métaéléments ayant des super-métaéléments.*

Figure 6. *Exemple : Formalisation d'un métamodèle* mm_{comp}

3.2. *Modèle : Définition*

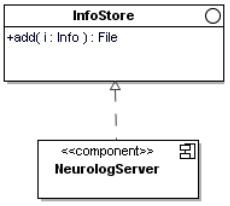
Un modèle est défini par un ensemble d'éléments de modélisation. Tout élément a une valeur qui est une liste ordonnée d'éléments du modèle pouvant être vide. A tout élément est associé un ensemble non vide de métaéléments. Nous notons M l'ensemble des modèles et E l'ensemble des éléments. La figure 7 donne un exemple de modèle.

Définition 3.2 (Modèle) *Un modèle* m *est défini comme un ensemble d'éléments appartenant à un ensemble de modélisation appartenant à un ensemble de métamodèles tel que* m=$\{e_1, \ldots, e_n\}$ *et tel que pour tout* $e \in m$ *:*

- *value(e) = [*e_i*,...] une liste ordonnée d'éléments pouvant être vide,*
- *metas(e)=* $\{me | me \in ME\}, |metas(e)| \geq 1$*,*
- *ident(e)=i,* $i \in ID$

La fonction $metas^* : E \rightarrow \mathscr{P}(ME)$ renvoie l'ensemble des métaéléments associés à un élément, y compris ceux définis par la relation *super* des métamodèles. Cette fonction est définie par :
$\forall e \in E, metas^*(e) = \{me | (me \in ME, me \in metas(e)) \vee (\exists me' \in metas(e),\ super(me') = me)\}$

Les métaéléments de l'exemple étant basés sur la métamodélisation d'un diagramme à composants (cf. figure 6), nous utilisons la visualisation et le vocabulaire des diagrammes à composants UML pour faciliter la compréhension.



m_0={NeurologServer,InfoStore,Ref-NeurologServer-InfoStore,Add,Ref-InfoStore-add,Info-param,Ref-add-Info, Infoparam,Ref-add-File,Fileparam}.

metas(NeurologServer)={Component} *metas*(InfoStore)={Interface}
metas(Add)={Operation} *metas*(Infoparam)=*metas*(Fileparam)={Parameter}
metas(Ref-NeurologServer-InfoStore)={Exposed} *value*(Ref-NeurologServer-InfoStore)=[NeurologServer,InfoStore]
metas(Ref-InfoStore-add)={OwnedOp} *value*(Ref-infoStore-add)=[InfoStore,Add]
metas(Ref-add-Info)={InParam} *value*(Ref-add-Info)=[Add,Info]
metas(Ref-add-File)={ReturnedParam} *value*(Ref-add-File)=[Add,File]

Figure 7. *Exemple : Formalisation du modèle initial* $m0$

Un modèle est donc défini par un ensemble d'éléments issus éventuellement de métamodèles différents.

Définition 3.3 (Assemblage) *La notion d'assemblage associe un modèle à l'ensemble des métamodèles auxquels il fait référence. Un assemblage A est défini par un couple (MM_{ref}, m) tel que :*

– $MM_{ref} \subseteq MM$

– $m \in M$, $\forall e \in m, \forall me \in metas(e), \exists mm \in MM_{ref}, me \in mm$

Un métaélément peut appartenir à plusieurs métamodèles ; nous ne garantissons pas la minimalité de l'assemblage.

3.3. *Conformité d'un modèle relativement à des métamodèles*

3.3.1. *Conformité dans une approche à métamodèle unique*

L'association d'un élément et d'un métaélément nous permet de définir la relation de conformité d'un élément en fonction des contraintes exprimées sur le métaélément. Les contraintes sont capturées relativement à un métamodèle dans le prédicat $isConform : MM \times ME \times E \rightarrow Boolean$.

Un modèle m est alors *conforme* relativement à un métamodèle mm si chacun de ses éléments référence un métaélément de mm et respecte le prédicat de conformité qui lui est associé[5].

Définition 3.4 (Conformité (simple) d'un modèle) *Un modèle m est conforme relativement à un métamodèle mm (noté $m \in mm$[6]) si :*
$\forall e \in m$, $mm \cap metas^*(e) \neq \emptyset \wedge \forall me \in mm \cap metas^*(e)$, $isConform(mm, me, e)$.

Un métamodèle mm définit donc la cohérence des modèles qui s'y réfèrent par un ensemble de conformités.

Par exemple, dans le métamodèle mm_{comp} (Fig. 6), au métaélément $exposed$ est associé le prédicat $isConform_{exposed}^{mm_{comp}}$ qui exprime qu'un élément ayant pour métaélément $exposed$ prend pour valeur un composant et une interface, contraintes exprimées ci-après en OCL :

```
context element:exposed
inv : self.value->([e1,e2] | e1.metas->includes(Component)
and e2.metas->includes(Interface))
```

Dans le modèle initial (cf. figure 7) le prédicat $isConform_{exposed}^{mm_{comp}}$(ref-NeurologServer-InfoStore) est vérifiée.
Il en est ainsi pour tous les éléments de $m0$ qui vérifient le prédicat de conformité associé aux métaéléments référencés, donc $m0 \in mm_{comp}$.

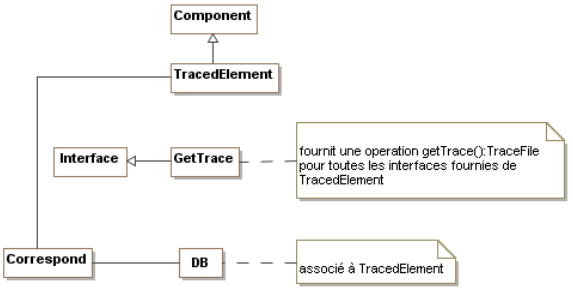
Figure 8. *Exemple : Cohérence de $m0$ relativement à mm_{comp}*

3.3.2. *Enrichissement d'un assemblage*

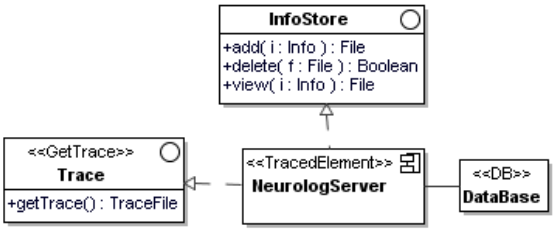
Basée sur cette formalisation, l'évolution d'un assemblage se fait par les actions suivantes : (i) l'ajout ou le retrait d'un élément, (ii) la mise à jour d'une valeur ou (iii) l'association d'un élément et d'un métaélément. Le métaélément ajouté fait alors partie soit d'un métamodèle déjà introduit dans l'assemblage, soit d'un métamodèle enrichissant l'assemblage. Ainsi un élément peut être associé à plusieurs métaéléments, appartenant à des métamodèles différents n'ayant pas obligatoirement de lien entre eux. Cette approche facilite la séparation des préoccupations en ne forçant pas le référencement à un métamodèle unique [STE 04]. La figure 9 illustre un enrichissement du modèle initial par un système de trace où un nouveau métaélément est introduit correspondant au concept de base de données qui n'était pas présent dans le métamodèle mm_{comp}.

5. Notre démarche ne définit pas de relation de typage, d'héritage ou d'instance. Nous n'avons besoin que de la relation $metas$ qui associe un élément à ses métaéléments. Les relations dont nous avons parlé précédemment ainsi que les conformités structurelles d'`UML` ou d'`OCL` peuvent être définies par le prédicat de conformité associé au métaélément. De plus d'autres techniques peuvent être utilisées comme l'écriture de programmes de tests [BÉZ 05] ou la transformation vers d'autres espaces technologiques où la vérification est plus simple [VIR 04, POU 07].
6. Cette notion a été choisie relativement à la notation dans le domaine des langages a été appliqué à l'IDM [BLA 06]

Le métamodèle mm_{trace} définit les métaéléments nécessaires à la politique de gestion des traces. Il est construit sur la base de relations existantes dans mm_{comp} qu'il spécialise et enrichit. Ainsi les éléments de métaélément `TracedElement` sont associés par une référence à un élément de métaélément `DB`, qui est nouveau, et doivent fournir une interface de métaélément `GetTrace`...
La modélisation ci-dessous illustre les métaéléments de mm_{trace} auxquels nous ajoutons des contraintes.Nous utilisons la relation d'héritage UML pour noter la relation super entre métaéléments.



Le modèle $m1_{trace}$ ci-dessous représente le modèle $m0$ enrichi d'un système de gestion de trace. Nous visualisons par des stéréotypes les relations entre les éléments et les métaéléments.



Le modèle $m1_{trace}$ n'est pas conforme à mm_{comp} par la présence de l'élément $DataBase$ associé au métaélément DB qui n'existe pas dans mm_{comp}. Cependant, $m1_{trace}$ est conforme à ce qui était attendu : une modélisation du système NeuroLOG avec gestion de la journalisation. Cela nous a conduit à introduire une relation de conformité relativement à un ensemble de métamodèles par restriction des éléments du modèle.

Figure 9. *Exemple : Enrichissement par l'ajout de trace ($m1_{trace}$)*

3.3.3. *Conformité dans une approche multi-métamodèles*

Le calcul des conformités relatives à chaque métamodèle demande alors d'isoler les éléments selon les métaéléments qui leurs sont associés. Afin de pouvoir extraire les éléments appartenant à un même métamodèle nous introduisons la notion de *restriction d'un modèle* à un métamodèle. Nous considérons que seuls les éléments conformes à un métaélément du métamodèle et dont toutes les valeurs appartiennent au métamodèle appartiennent à la restriction du modèle.

Définition 3.5 (Restriction d'un modèle relativement à un métamodèle) *Nous notons $m_{/mm}$ la restriction d'un modèle m à un métamodèle mm telle que :*
$m_{/mm} = \{e_i \in m | \exists me \in mm \cap metas^*(e_i) \wedge \forall x \in value(e_i), \exists me \in mm \cap metas^*(x)\}$

Cette définition sert alors de base pour définir la conformité d'un modèle relativement à un métamodèle dans un assemblage. Chaque élément de la restriction du modèle doit alors être conforme à ses métaéléments ce qui s'énonce comme suit.

Définition 3.6 (Conformité d'un modèle dans un assemblage) *Soit $A = (MM_{ref}, m)$ un assemblage. Le modèle m est conforme à $mm \in MM_{ref}$ si $m_{/mm} \in mm$.*

Un assemblage est alors *cohérent* si le modèle est conforme à l'ensemble des métamodèles MM_{ref}. La conformité est alors définie par la conformité à chaque métamodèle.

Définition 3.7 (Cohérence d'un assemblage) *Soit* $A = (MM_{ref}, m)$ *un assemblage tel que* $MM_{ref} = \{mm_1 ... mm_n\}$. *A est cohérent si* $\forall mm \in MM_{ref}$, $m_{/mm} \in mm$

La figure 10 illustre la restriction de $m1_{trace}$ relativement aux métamodèles mm_{comp} et mm_{trace}. Chacun des modèles obtenu est alors conforme au métamodèle référant.

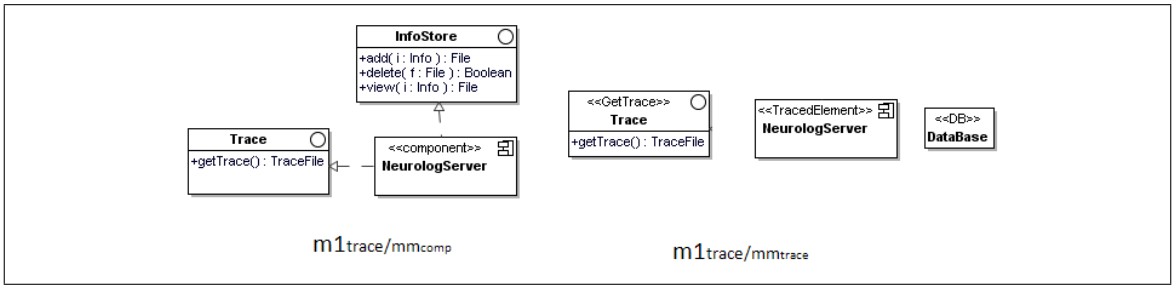


Figure 10. *Restriction de* $m1_{trace}$ *par rapport à* mm_{comp} *et* mm_{trace}

En définissant la restriction d'un modèle relativement à un métamodèle, nous avons formalisé la cohérence d'un modèle relativement à plusieurs métamodèles. Nous utilisons désormais cette définition de conformité. De la même manière, lorsque nous parlons de métamodèles ou de modèles, nous référençons les définitions posées dans cette section. La définition de la conformité relative à plusieurs métamodèles nous sert alors de base pour étudier la conformité d'un modèle construit par introduction de politiques. La section suivante étudie l'évolution d'un modèle en formalisant les politiques sous forme de transformations de modèles idempotentes.

4. Transformations idempotentes

Notre objectif est de permettre de réparer automatiquement un modèle pour qu'il respecte toutes les contraintes portées par les politiques introduites. Nous définissons les politiques comme des transformations idempotentes. Pour cela nous formalisons la notion de transformation et d'application (voir 4.1.1) auxquelles nous ajoutons des propriétés section 4.1.2 afin de permettre la ré-application des transformations.

4.1. *Transformations de modèles*

4.1.1. *Notions générales*

Une transformation est définie par le quadruplet (mm_{source}, mm_{cible}, $Expr$, S_{free}). Le métamodèle source définit la conformité attendue par le modèle à transformer. Le métamodèle cible définit la conformité attendue par les modèles résultant de la transformation. Enfin $Expr$ est l'expression de la transformation qui fait référence à un ensemble de variables S_{free} typées. Ces variables sont utilisées pour lier la transformation au modèle. Les variables correspondent à des rôles que jouent les éléments du modèle lors de l'application de la transformation.

La liaison d'une transformation τ à un modèle m définit une substitution entre les variables de la transformation et des termes clos correspondant soit à des éléments du modèle, soit à des identifiants [SAD 97]. Une liaison est *complète* si toutes les variables libres de S_{free} sont substituées à un élément du modèle ou à un identifiant. On note $\sum_{\sigma}^{\tau}$ l'ensemble des liaisons complètes pour une transformation τ. On appelle alors **transformation contextualisée** une transformation associée à une liaison complète.

L'application d'une transformation contextualisée (τ, $\sigma_{\tau,m}$) n'est possible que si (i) l'application est contextualisée relativement au modèle m et (ii) le modèle m est conforme au métamodèle source. On note alors $\tau_{\sigma}(m) = m'$ l'application de la transformation τ sur le modèle m selon la liaison complète $\sigma_{(\tau,m)}$. Le modèle résultant m' est alors conforme par construction au métamodèle cible, $m'/mm_{cible} \in mm_{cible}$. Nous considérons ce point toujours vrai. Différents travaux sur la validation des transformations étayent ce postulat [BRO 06].

L'ajout de la politique P_{trace} peut être automatisé par une transformation. Nous notons cette transformation $\tau trace$ = ($mm_{tracesource}$, $mm_{tracecible}$, $Expr_{trace}$, S_{trace}) où :

– $mm_{tracesource}$= mm_{comp} représenté figure 6

– $mm_{tracecible}$ = mm_{trace} représenté figure 9

– S_{trace} = $\{tracedElement\}$

– $Expr_{trace}$ L'expression dépend du langage de transformation utilisé. Ici nous présentons sous forme littérale cette transformation : *L'élément de rôle* $tracedElement$ *expose une interface de journalisation qui offre une méthode permettant d'obtenir les fichiers de journalisation. Les fichiers de journalisation sont gérés par une base de données reliée à l'élément de rôle* $tracedElement$.

L'application de cette transformation sur le modèle initial vise à améliorer le suivi des accès au serveur. La liaison de la transformation est alors définie de la manière suivante $\sigma_{(\tau trace,m0)} = \{(tracedElement \rightarrow NeurologServer)\}$. Comme le modèle initial $m0$ est conforme au métamodèle source de la transformation on obtient $\tau trace_{\sigma}(m0) = m1_{trace}$. Le résultat est illustré figure 9.

Figure 11. *Exemple : Transformation contextualisée pour l'introduction de la gestion de trace*

4.1.2. *Des transformations idempotentes support aux politiques*

L'objectif est d'utiliser le caractère constructif de la transformation pour réparer les modèles et ainsi maintenir un modèle conforme à une politique en ré-appliquant une transformation. Seule une transformation contextualisée ayant la propriété d'idempotence peut être appliquée plusieurs fois en créant des modèles résultants conformes au métamodèle cible. Nous notons T_i l'ensemble des transformations idempotentes.

Définition 4.1 (Transformation idempotente) *Soit* $\tau \in T$ *telle que* τ=(mm_{source},mm_{cible},$Expr$,S_{free}) *τ est idempotente si :* $\forall\ m \in mm_{source}$ *et* $\forall\ \sigma_{(\tau,m)} \in \sum_{\sigma}^{\tau}$, $\tau_{\sigma}(m) = m' \Rightarrow \tau_{\sigma}(m') = m'' \equiv m'$[7]

En corollaire à cette définition deux propriétés sont induites par l'idempotence : (Prop 1) pour pouvoir se ré-appliquer, le modèle résultant d'une application doit aussi être conforme au métamodèle source de la transformation. Il y a donc inclusion des métamodèles cible et source[8]. On peut alors parler de transformation endogène par restriction ; (Prop 2) pour pouvoir se ré-appliquer il faut que la liaison de la transformation soit toujours complète.

Chaque politique enrichit le modèle en ajoutant des éléments correspondant à son métamodèle cible. Lorsque plusieurs politiques sont introduites, l'ensemble des métamodèles cibles et le modèle constituent alors un assemblage que l'on veut cohérent (cf. definition 3.7). Une même politique peut être introduite sur un modèle en utilisant différentes liaisons complètes. Ce point nous conduit à définir la notion de Système qui étend l'assemblage en lui ajoutant les politiques qui doivent être introduites sur le modèle.

Définition 4.2 (Système cohérent) *Soit* $S = (P_{context}, MM_{ref}, m)$ *un système tel que :*

– $P_{context} = \{Pi_{(\sigma,m)}\}$ *un ensemble de transformations contextualisées,*

– MM_{ref} *un ensemble de métamodèles, tel que* $\forall Pi, mmi_{cible} \in MM_{ref}$

– m *un modèle*

S est cohérent si, $\forall mm \in MM_{ref}, m_{/mm} \in mm$.

Notre objectif est de construire un système cohérent selon le processus de modélisation détaillé dans la section suivante.

7. Nous considérons que la liaison s'applique sur les éléments correspondants dans le modèle résultat.
8. Soient $mm_1, mm_2 \in MM^2$. Le métamodèle mm_1 est inclus dans le métamodèle mm_2 (noté $mm_1 \subset mm_2$) si $m \in mm_1 \Rightarrow m \in mm_2$

4.2. *Des transformations idempotentes par construction*

Afin d'établir les fondements d'un système de transformations qui supporte la réparation de modèles, nous définissons des transformations idempotentes par construction (ITC) [NEM 10b] sur la base d'un jeu d'actions élémentaires.

4.2.1. *Langage d'actions élémentaires*

Relativement au métamodèle défini à la section 3, nous définissons un ensemble d'actions élémentaires permettant de sélectionner des éléments du modèle, d'identifier des éléments et de modifier des éléments du modèle. Chacune de ces actions a pour pré-condition que les éléments en paramètre existent.

Actions de sélection :

metas est l'action qui pour un élément de modèle renvoie son (ou ses) métaéléments. Par construction, à un élément est associé au moins un métaélément : $|metas(m, e)| \geq 1$.

value est l'action qui pour un élément du modèle renvoie sa valeur sous la forme d'un ensemble de suites d'éléments (éventuellement vide). Par construction, à un élément donné est associé au plus une valeur : $|value(m, e)| \leq 1$

ident est l'action qui détermine l'ensemble des identifiants associés à un élément d'un modèle ou d'un métamodèle. Par construction, à un élément (ou métaélément) e donné d'un modèle ou métamodèle m est associé exactement à un identifiant : $|ident(m, e)| = 1$

hasForMeta est l'action qui pour un élément du métamodèle renvoie tous les éléments de modèle dont il est le métaélément :
$hasForMeta(m, meta) = \{e \in m | meta \in metas^*(m, e)\}$

hasForValue est l'action qui sélectionne les éléments de modèle de valeur donnée :
$hasForValue(m, [e_1...e_n]) = \{e \in m | value(m, e) = [e_1...e_n]\}$

hasForIdent est l'action qui sélectionne les éléments ou métaéléments ayant un identifiant donné. Par construction, il existe au plus un élément correspondant à un identifiant donné :
$hasForIdent(m, id) = \{e \in m | ident(m, e) = id\}$

Actions d'identification : Une action d'identification détermine pour une suite d'identifiants donnée, un identifiant unique. Cette action **getUniqueKey** permet de caractériser de manière unique les éléments. Elle doit donc être injective, i.e. que pour un identifiant donné, il existe au plus une suite d'identifiants qui est son antécédent. Par exemple, pour le métamodèle à composant (cf. exemple 6) défini précédemment, une opération est définie de manière unique par son nom et l'interface à laquelle elle est associée.

Actions de modification : Dans les travaux sur la gestion de la cohérence structurelle [FOM 07] les actions de modification d'un modèle sont basées sur la définition d'un modèle comme ensemble d'éléments, de propriétés et de références. Les actions autorisées sont alors la création d'un élément, la modification d'une propriété, la valuation d'une référence et la suppression d'un élément. Dans notre démarche nous considérons les propriétés et les références comme des éléments de modélisation que l'on peut valuer et auxquels on associe des métaéléments particuliers. Nous ne les distinguons pas. Les actions de création, valuation et suppression des éléments permettent alors de manipuler les éléments d'un modèle. Nous ajoutons l'action d'ajout d'un métaélément afin de permettre la référence de plusieurs métaéléments par un élément, ce qui est particulier à notre démarche. Nous définissons alors quatre actions de modification

createElement est l'action qui crée un élément d'identifiant id donné associé à un métaélément $meta$ donné dans un modèle m.

setValue est l'action qui modifie la valeur d'un élément donné.

deleteElement est l'action qui détruit un élément du modèle et met à nul toutes les valeurs qui font référence à cet élément.

setMeta est l'action qui ajoute une association vers un métaélément pour un élément donné.

L'ensemble de ces actions de modification sont **idempotentes**. Cette propriété nous permet d'appliquer plusieurs fois la même action sur un modèle sans le modifier[9].

9. La démonstration de cette propriété et le détail des impacts de ces actions sur le modèle se trouvent p. 115 du document [NEM 10a]

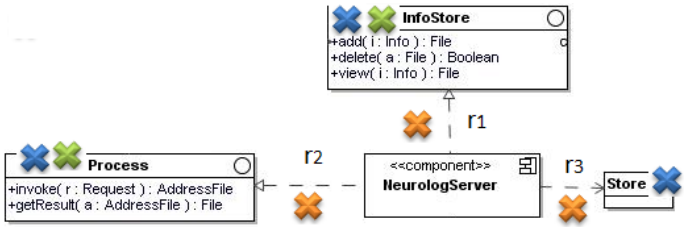
4.2.2. *Évaluation des actions*

Nous avons modélisé l'évaluation des actions initialement sous forme de fonctions paramétrées par une représentation en logique. Nous définissons un **appel d'actions** comme un tuple *(terme,VarsIn,VarOut)* où le *terme* désigne une action par son nom (e.g. `hasForMeta, setValue`), `VarsIn` les arguments d'appel de l'action et `VarOut` un des résultats possibles de l'appel. Les arguments d'appel sont désignés par une suite de termes qui peuvent correspondre à des variables, des identifiants ou des tuples de termes. Nous notons AA l'ensemble des appels d'actions. La figure 12 donne des exemples d'appels d'actions.

L'**évaluation d'un appel d'action** (notée $eval$) renvoie toutes les substitutions satisfaisant le prédicat correspondant à l'appel. Notons que si l'ensemble des éléments résultats est vide l'évaluation retourne un ensemble vide de substitutions. La figure 12 exemplifie l'évaluation d'une suite d'appels d'actions.

Les appels d'actions suivants ont pour but de sélectionner toutes les interfaces exposées par un composant C.
Renvoie dans I tous les éléments de modèles qui ont interface *pour métaélément :*
aa1 = (hasForMeta,[M,interface],I)
Renvoie dans R les éléments de modèle ayant pour valeur [I,C] :
aa2 = (hasForValue,[M,[I,C]],R)
Renvoie dans R les éléments de modèles qui ont exposed *pour métaélément :*
aa3 = (hasForMeta,[M,exposed],R)
L'utilisation d'une même variable R dans deux appels d'actions permet de construire l'intersection des ensembles. Ici il s'agit de trouver les références `R` de métaélément `exposed` entre les éléments `I` et `C`.

La suite d'appels d'actions $[aa1, aa2, aa3]$ est liée au modèle m par la liaison :
$\sigma = \{(M \rightarrow m), (C \rightarrow neurologServer.$



Le schéma montre par une croix bleu (respectivement orange et vert) les valeurs de retour de l'appel d'action aa1 (respectivement aa2 et aa3). L'évaluation de la suite des appels d'actions construit les ensembles suivants :
eval(aa1,σ) = {
$\{(M \rightarrow m), (C \rightarrow neurologServer), (I \rightarrow infoStore)\}$
$\{(M \rightarrow m), (C \rightarrow neurologServer), (I \rightarrow process)\}$
$\{(M \rightarrow m), (C \rightarrow neurologServer), (I \rightarrow store)\}\}$

eval([aa1,aa2],σ) = {
$\{(M \rightarrow m), (C \rightarrow neurologServer), (I \rightarrow infoStore), (R \rightarrow r1)\}$
$\{(M \rightarrow m), (C \rightarrow neurologServer), (I \rightarrow process), (R \rightarrow r2)\}$
$\{(M \rightarrow m), (C \rightarrow neurologServer), (I \rightarrow store), (R \rightarrow r3)\}\}$

eval([aa1,aa2,aa3],σ) = {
$\{(M \rightarrow m), (C \rightarrow neurologServer), (I \rightarrow infoStore), (R \rightarrow r1)\}$
$\{(M \rightarrow m), (C \rightarrow neurologServer), (I \rightarrow process), (R \rightarrow r2)\}\}$

Figure 12. *Exemple : Evaluation d'une suite d'appels d'actions*

4.2.3. *Unification des appels d'actions de modification*

L'application d'une transformation consiste à évaluer les suites d'actions. Dans le cas des actions de modifications, la détection des conflits et le ré-ordonnancement doit être faits avant leurs évaluations et en fonction des substitutions des évaluations des actions de sélection et d'identification. L'objectif est donc par

unification de calculer l'ensemble des suites d'actions de modification à évaluer pour pouvoir les analyser et les composer. On appelle **appel d'action unifié** un appel d'action de modification où les variables d'entrée ont été substituées et la fonction d'unification est notée $unify$.

Exemple : Unification d'un appel d'action
$unify((setValue, [M, E, V], M1), \{M \rightarrow m, E \rightarrow e, V \rightarrow [v1, v2]\}) = (setValue, [m, e, [v1, v2]], M1)$

Par extension nous définissons l'unification d'une suite d'appels d'actions de modification par l'application de la substitution à chacun des appels de la suite. Puis, nous généralisons les définitions précédentes dans le cas d'une suite d'appels d'actions et d'un ensemble de substitution. On note AA^n l'ensemble des suites à n éléments.

unifys : $AA^n \times \mathscr{P}(\Sigma) \rightarrow \mathscr{P}(AA^n)$

$unifys([a1...an], \{\sigma_1 \dots \sigma_m\}) = \{unify([a1...an], \sigma_1) \dots unify([a1...an], \sigma_m)\}$

La figure 13 visualise l'unification d'une suite d'actions relativement à un ensemble de substitutions.

La suite d'appels d'actions suivante permet d'ajouter une opération à l'interface I.
aa6 = (createElement,[M,ID,operation],M1) *construit un modèle M1 où l'élément d'identifiant ID et de métaéléments opération existe*
aa7 = (createElement,[M1,IDRef,ownedOp],M2) *construit un modèle M2 où l'élément d'identifiant IDRef et de métaélément ownedOp existe*
aa8 = (setValue,[M2,IDRef,[I,ID]],M3) *construit un modèle où l'élément d'identifiant IDRef a pour valeur [I,ID]*

L'unification est faite selon le retour des évaluations de la suite d'actions de sélection donnée en exemple dans la section 4.2.2 et du retour de l'évaluation d'une suite d'actions d'identification. L'ensemble des substitutions formées est alors :
σ_1=$\{(M \rightarrow m), (C \rightarrow NeurologServer), (I \rightarrow infoStore), (R \rightarrow r1), (ID \rightarrow id1), (IDRef \rightarrow id2)\}$
σ_2=$\{(M \rightarrow m), (C \rightarrow NeurologServer), (I \rightarrow process), (R \rightarrow r2)(ID \rightarrow id3), (IDRef \rightarrow id4)\}$

L'unification de la suite d'appels d'actions [aa10, aa11, aa12] avec l'ensemble des substitutions $\{\sigma_1, \sigma_2\}$ donne l'ensemble des appels d'actions suivants :
unifys([aa6,aa7,aa8],$\{\sigma_1, \sigma_2\}$)
= {[(createElement,[m,id1,operation],M1),(createElemenmt,[m,id2,ownedOp],M2),
(setValue,[m,id2,[infoStore,id1]],M3)],
[(createElement,[m,id3,operation],M1),(createElement,[m,id4,ownedOp],M2),
(setValue,[m,id4,[process,id3]],M3)],}

Figure 13. *Exemple : Unification d'une suite d'appels d'actions pour un ensemble de substitution*

4.2.4. *ITC : écriture*

Nous appelons **transformation idempotente par construction (ITC)**, une transformation pour laquelle l'idempotence est garantie par la vérification de contraintes lors de l'écriture. Nous appelons T_{itc} l'ensemble de ces transformations. La figure 14 présente un exemple d'ITC simple.

Définition 4.3 (ITC) *Soit $\tau = (mm_{source}, mm_{cible}, Expr, S_{free}) \in T_i$, τ est une transformation idempotente par construction (ITC) si elle respecte les contraintes suivantes*[10].

– *Expr = (Select, Ident, Modif), avec* $(Select, Ident, Modif) \in \mathscr{P}(AA) \times \mathscr{P}(AA) \times \mathscr{P}(AA)$,

– C1 : Il n'existe pas d'action de modification qui détruise un élément de liaison.
$\forall (deleteElement, [M, E], M') \in Modif,\ E \notin S_{free}$

– C2 : L'action de sélection ne peut se faire que sur des métaéléments du métamodèle cible.
$\forall hasForMeta(m, meta) \in Select \Rightarrow meta \notin (mm_{cible} \setminus mm_{source})$[11]

10. Pour des raisons de place nous omettons les contraintes structurelles (identification unique du modèle source, enchaînement des appels d'actions, clôture des appels aux actions de modification), pour nous focaliser sur les contraintes sur les actions.
11. La différence entre deux métamodèles $mm1$ et $mm2$, notée $mm1 \setminus mm2$ est définie comme :
$mm1 \setminus mm2 = \{me \in mm1 | (me \notin mm2) \wedge (\forall x \in super(me), x \notin mm2)\}$.

– C3 : L'action d'ajout d'une association vers un métaélément ne peut se faire que sur un métaélément strictement inclus dans le métamodèle cible .
$\forall setMeta(m, id, meta) \in Expr \Rightarrow meta \in (mm_{cible} \setminus mm_{source})$.
En effet, nous considérons que si une politique doit qualifier plus précisément un élément, elle doit le faire dans le contexte même de la transformation, contexte que nous capturons par les métaéléments du métamodèle cible.

Soit une ITC qui ajoute une opération à toutes les interfaces fournies par un composant C.

Select = $[aa1 = (hasForMeta, [M, interface], I),$
$aa2 = (hasForValue, [M, [C, I]], R),$
$aa3 = (hasForMeta, [M, exposed], R)]$

Ident = $[aa4 = (getUniqueKey, [C, I, operation], ID),$
$aa5 = (getUniqueKey, [C, I, ownedOp], IDRef)]$

Modif = $[aa6 = (createElement, [M, ID, operation], M1)$
$aa7 = (createElement, [M1, IDRef, ownedOp], M2),$
$aa8 = (setValue, [M2, IDRef, [I, ID]], M3)]$

$S_{free} = \{M, C\}$ l'ensemble des variables libres de l'ITC.

Figure 14. *Exemple : Expression d'une ITC*

L'écriture des ITCs ne suffit pas pour garantir la préservation de la conformité source et la préservation des liaisons (définies dans la section 4.1.2). C'est le moteur d'application des ITCs qui prend en charge le respect de ces propriétés. Nous le présentons à présent.

5. Un processus d'évaluation supportant la réparation

Pour assurer l'idempotence des transformations, l'évaluation d'un appel d'action doit dépendre du contexte de l'évaluation.

En effet, une transformation peut créer de nouveaux éléments, modifier les valeurs des éléments ou encore ajouter des références vers des métaéléments. Ces modifications entraînent alors un changement potentiel des éléments sélectionnés, en particulier les éléments valués ou créés par la transformation, et entraîne alors de nouvelles modifications, non voulues. Le processus de transformation doit donc interdire la sélection des éléments créés ou valués par l'application d'une transformation contextualisée lors de la ré-application de cette transformation.

De plus si l'utilisateur ou l'application d'une ITC détruit (cas 1) des éléments de liaison ou (cas 2) des éléments qui ont servi à la création d'éléments par l'application d'ITC, nous ne savons pas automatiquement rétablir la cohérence d'un Système. En effet, dans le premier cas, la transformation ne peut pas être ré-appliquée. Dans le second cas, la création/destruction d'un élément ne peut pas être exprimée sur la non existence d'éléments (nous ne l'autorisons pas), du coup nous ne savons pas détruire les éléments devenus inutiles. Pour éviter ces situations, nous interdisons de supprimer des éléments faisant partie des liaisons définies pour l'application des ITCs du Système (pas seulement ceux de la liaison de la transformation en cours d'exécution) ou des éléments utilisés dans la sélection d'autres ITCs[12]. Le processus d'évaluation des transformations décrit dans cette section le contrôle donc les tentatives de violation de ces règles.

Nous avons fait le choix de gérer le **contexte d'application** d'une ITC par le moteur d'application (cf. §5.1) et de définir l'application d'une ITC en prenant en compte ce contexte pour restreindre la sélection des éléments et interdire des modifications qui empêcheraient la réparation des modèles (cf. §5.2).

12. Notons que si des destructions d'éléments sont exigées, les politiques correspondantes doivent être "retirées" par l'utilisateur du système. Nous utilisons dans ce cas une opération prédéfinie de retrait de politique qui utilise le contexte [NEM 10a](page 121).

5.1. *Contexte d'application*

A un couple (τ, σ) correspondant à une ITC contextualisée, la fonction *getUniqueIdentContext* associe un identifiant unique, dit *identifiant de contexte*. Nous utilisons le terme de contexte pour alléger le discours lorsqu'il n'y a pas d'ambiguïté. Nous notons $ID_{context}$ l'ensemble des identifiants de contexte. A une action utilisateur, nous associons un identifiant unique que nous notons $user$.

Le contexte d'application nous permet de tracer les actions opérées sur le modèle. Ainsi la fonction *context* renvoie pour chaque élément du modèle un ensemble d'identifiants de contexte en fonction des actions élémentaires.

context : $Terme \times E \times M \rightarrow \mathscr{P}(ID_{context})$
context(meta,e,m) = $\{id_{ap_1}..id_{ap_n}\}$ *: contextes qui ont associés un métaélément à e.*
context(value,e,m) = $\{id_{ap}\}$ *: contexte de la dernière application ou action utilisateur qui a valué e.*
context(ident,e,m) = $\{id_{ap_1}..id_{ap_n}\}$ *: contextes qui ont créés e ou ont demandés sa création.*
context(delete,e,m) = $\{id_{ap_1}..id_{ap_n}\}$ *: contextes qui ont demandés la suppression de e.*
context(select,e,m) = $\{id_{ap_1}..id_{ap_n}\}$ *: contextes correspondant à l'évaluation des actions* $hasForMeta$, $hasForValue$, $hasForIdent$ *qui ont sélectionnés* e.
context(getValue,e,m) = $\{id_{ap_1}..id_{ap_n}\}$ *: contextes à l'évaluation de l'action* $value$*, qui ont sélectionné la valeur de* e.

5.2. *Evaluation contextualisée d'une suite d'appels d'action*

L'évaluation contextualisée des appels d'actions joue deux rôles. D'une part, elle restreint l'ensemble des éléments sélectionnés à ceux qui n'ont pas été créés ou valués par une précédente application de cette transformation contextualisée (pour une même transformation et une même liaison). D'autre part, elle met à jour le contexte des éléments pour les éléments créés ou modifiés par les actions de modification.

Nous appelons $evals_{context}$ la fonction d'évaluation contextualisée qui renvoie toutes les substitutions correspondant à l'évaluation de l'ITC sauf celles dont les éléments ont été valués ou créés par ce contexte. L'application contextualisée des actions de sélection ne prend pas en compte dans les substitutions les éléments créés ou valués par l'application[13]. Elle est définie par la fonction suivante :
$eval_{context} : AA \times \Sigma \times ID_{context} \rightarrow \mathscr{P}(\Sigma)$
$eval_{context}((T, vars, res), \sigma, id_c) = \{\sigma_1 \dots \sigma_n\}$
$= eval((T, vars, res), \sigma) \setminus \{\sigma_i | (\exists (res \rightarrow e) \in \sigma_i, (id_c \neq user) \wedge (id_c \in context(ident, e, m) \vee context(value, e) = \{id_c\})\}$
Nous généralisons l'évaluation contextualisée d'une action à un ensemble d'appels d'action par la fonction $evals_{context}$.

L'évaluation des actions de sélection et de modification met à jour le contexte. Nous mettons en avant dans le cadre de cet article uniquement les points essentiels de cette mise à jour.

Préservation du contexte : Le contexte des éléments e *non détruits ou valués* par l'évaluation est préservé au sens où les nouveaux contextes sont inclus :
$eval_{context}((A, VarIn, M), \sigma, ap) = \{M \rightarrow m', \dots\}, A \neq deleteElement, A \neq setValue$
$\forall e' \in m'$, si $\exists e \in m, e \approx e'$[14] $context(T, e', m') \supseteq context(T, e, m)$

Évolution du contexte en cas de valuation d'un élément : L'évaluation d'une action de valuation met à jour le contexte, si la valuation est autorisée. En effet, lorsque la valeur de l'élément a déjà été utilisée dans l'application d'une ITC, la réévaluation de l'élément est interdite.
$eval_{context}((setValue, [m, e, value], M), \sigma, ap) = \{M \rightarrow m', \dots\}$ et
Si $\nexists ap_i \in context(getValue, e, m)$
$context(value, e', m') = \{ap\}$ où $e' \in m', e \approx e'$
sinon $eval_{context}((setValue, [m, e, value], M), \sigma, ap)$ échoue et le modèle n'est pas modifié.

13. Notons qu'un utilisateur peut appliquer une action sur des éléments qu'il a créés ou valués.
14. $\approx$ définit l'équivalence des éléments au sens où ils ont les mêmes identifiants.

Évolution du contexte en cas de destruction d'un élément : Dans le cas de la destruction d'un élément, le contexte précédent est modifié pour refléter les changements de valeurs des éléments modifiés et enregistrer la destruction de l'élément. Cependant la destruction d'un élément apparaissant dans une liaison ou ayant été utilisé dans la partie sélection des ITCs ou ayant été sélectionné pour sa valeur est interdite.
$eval_{context}((deleteElement, [m, e], M), \sigma, ap) = \{M \rightarrow m', \ldots\}$
Si $\nexists ap_i \in context(select, e, m)$ et $\nexists ap_i \in context(getValue, e, m), ap_i \neq ap$
$context(delete, e, m') = \{ap\}$
$\wedge context(meta, e, m') = context(value, e, m') = context(ident, e, m') = \emptyset$
$\wedge \forall e_i \in m, \exists e'_i \in m', (e_i \approx e'_i) \wedge value(m, e_i) \neq value(m', e'_i)$
$\Rightarrow context(value, e'_i, m') = \{ap\}$
sinon $eval_{context}((deleteElement, [m, e], M), \sigma, ap)$ échoue et le modèle n'est pas modifié.

5.3. *Application d'une ITC*

Afin de garantir l'idempotence, l'application d'une ITC procède en 3 étapes :
Soient $\tau = (mm_{source}, mm_{cible}, (Select, Ident, Modif), S_{free}) \in T_{itc}$, et σ une liaison complète définie sur τ.

– **Étape 1 : Evaluation partielle** de l'ITC qui consiste en une évaluation contextualisée des parties *sélection* et *identification* de l'ITC puis *unification* de la partie modification de l'ITC :
$eval_{partial}(\tau, \sigma) = unifys(Modif, evals_{context}(Ident, evals_{context}(Select, \sigma, id_c), id_c)))$
avec $id_c = getUniqueIdentContext(\tau, \sigma)$

Les listes d'actions de modification obtenues en retour de l'évaluation partielle ne contiennent en variables libres que les variables désignant le modèle modifié. En effet toutes les variables d'entrée sont soient liées par la liaison (donc unifiées), soient liées aux variables de retour d'une action de sélection ou d'identification (donc unifiées grâce aux substitutions obtenues par évaluation contextualisée des actions de sélection et identification), soient liées aux variables de retour d'une action de modification précédente et qui désignent par définition le modèle modifié.

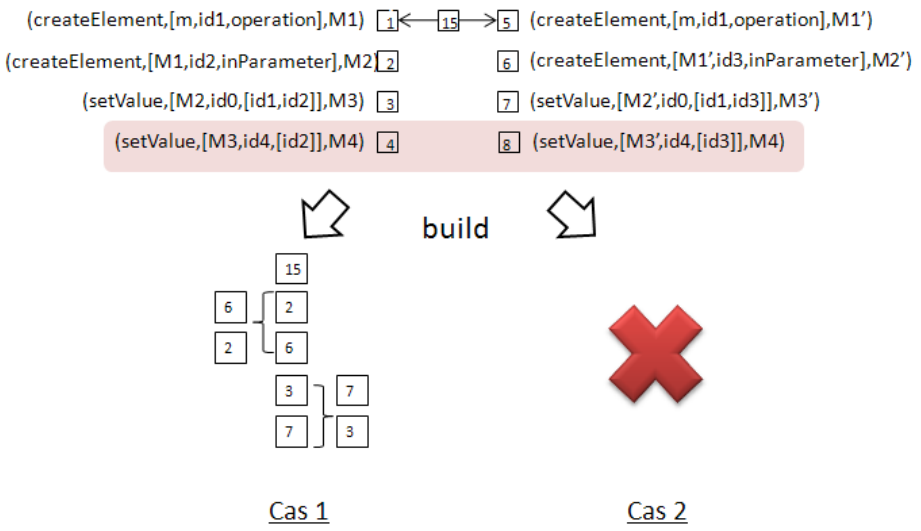
– **Étape 2 : Détection des conflits et construction d'une unique suite d'actions** :
L'évaluation partielle construit un ensemble de suites non forcément disjointes d'appels d'actions opérant potentiellement sur un même modèle. Par construction, nous avons l'hypothèse que chaque suite construit un modèle conforme au métamodèle cible. Dans un souci de déterminisme des évaluations que nous voulons indépendantes de l'ordre d'obtention des substitutions, cette étape consiste à construire à partir d'un ensemble de suites d'appels d'actions, une unique suite d'appels d'actions. Le processus de construction de cette unique suite procède en trois étapes : 1) unification des appels d'actions équivalentes (qui font appel à une même action et mettent en jeu les mêmes termes et les mêmes variables d'entrée et de sortie), 2) détection des conflits entre les appels d'action (détaillée ci-après), 3) en l'absence de conflits, production d'une suite ordonnée d'appels d'action. Dans le cas contraire le processus est interrompu.

Il y a un potentiel conflit entre deux suites d'appels d'actions partiellement ordonnées si nous trouvons des appels d'actions s'impactant mutuellement non ordonnés :
- $(delete, [m, e], M1)$ entre en conflit avec les appels suivants :
$(create, [\_, e, meta], \_)$; $(setMeta, [\_, e, meta], \_)$;
$(setValue, [\_, e, value], \_)$; $(setValue, [\_, e2, value], \_), e \in value$.
- $(setValue, [\_, e, Value], \_)$ entre en conflit avec
$(setValue, [\_, e, Value2], \_)$ si $Value \neq Value2$. Dans ce cas, un même élément peut prendre deux valeurs différentes selon l'ordre d'exécution des actions, il y a donc conflit. Notons que dans le cas où la valeur est la même, la première étape aura unifiée les deux actions.

Nous notons **build(**$\{[a_1..a_n] \ldots [b_1...b_m]\}$**)** la fonction qui prend en entrée un ensemble de listes d'appels d'actions et renvoie une liste d'appels d'actions construite par cet algorithme lorsqu'aucun conflit n'est détecté.

La figure 15 illustre la construction d'une suite d'appels d'actions.

– **Étape 3 : Evaluation de la suite d'actions de modification unifiée** obtenue à l'étape précédente.



Cas 1 (sans prendre en compte les actions en rouge) : Les appels d'actions 1 et 5 sont unifiés et remplacés par une action 15. Sans conflits détectés, la suite résultante respecte l'ordre partiel établit.
Cas 2 (toutes les actions sont prises en compte) : Les appels d'actions 1 et 5 sont unifiés et remplacés par les actions 15. Des conflits sont détectés entre les actions 4 et 8. Aucune suite n'est calculée.

Figure 15. *Etape 2 : Composition d'une unique suite d'appels d'actions de modifications*

Idempotence d'une ITC

Nous avons démontré dans [NEM 10a] (page 115) que l'application d'une ITC contextualisée est bien idempotente, i.e. :
Soit m un modèle, τ une ITC, σ et σ' deux liaisons complètes équivalentes[15] définies respectivement sur m et sur $m' = \tau_\sigma(m)$, Il y a idempotence si $\tau_{\sigma'}(m') = m'' \Rightarrow m' \equiv m''$

Cette propriété a été démontrée en prouvant qu'aucune nouvelle substitution n'est produite lors de l'application de τ à m'. En effet, si cette propriété est vérifiée les suites d'appels de modification produites sont des sous ensembles des suites d'appels produites lors de la précédente application de la transformation (opérations unifys). La composition de ces suites donne alors une sous suite des appels d'action qui lors de l'évaluation construit un modèle équivalent puisque chaque action de modification est idempotente.

6. Stratégies d'applications des ITCs et réparation du modèle

Dans une démarche d'évolution incrémentale des modèles [BAT 04], le modèle évolue à la fois par des actions utilisateur et par l'application des transformations. Assurer la cohérence du système revient dans notre approche à réparer le modèle en appliquant l'ensemble des ITCs. Ainsi nous proposons une approche mixte de la construction des systèmes qui supporte une introduction séquentielle des ITCs et la réparation par une approche ensembliste. L'utilisateur peut aussi faire le choix de n'utiliser que cette dernière approche, surtout s'il n'a pas d'intérêt à manipuler les éléments non métiers introduits par les politiques ou s'il souhaite utiliser uniquement le résultat final. Ainsi nous ne forçons pas la donnée d'un ordre sur l'application des transformations mais s'il est nécessaire il peut être appliqué.

15. $\forall(V \rightarrow e) \in \sigma, \exists(V \rightarrow e) \in \sigma'$tel que $e \approx e'$ ou e=e'(même terme) ou si $e = [e_1..e_n]$ et $e' = [e'_1..e'_n]$ alors $\forall e_i \in e, e'_i \in e_i \approx e'_i$ ou $e_i = e'_i$

6.1. *Approche séquentielle*

L'approche séquentielle est une stratégie d'application où les éléments ajoutés au modèle sont accessibles aux actions utilisateurs ou à la liaison avec d'autres politiques. Ainsi le modèle résultant de chaque application est utilisé pour appliquer une nouvelle transformation. L'ordre d'application des transformations joue alors un rôle important. Les transformations qui s'appliquent sur des éléments créés par d'autres politiques ne peuvent pas s'appliquer sans que celles-ci aient été appliquées. Avec cette stratégie d'application, le Système évolue incrémentalement par actions de l'utilisateur et applications d'ITCs. Réparer un modèle revient alors à appliquer toutes les transformations dans une approche ensembliste (cf. section 6.2). Si le modèle est modifié l'ensemble des transformations est ré-appliqué jusqu'à atteindre un modèle qui n'est plus modifié par application des transformations ou qu'un cycle soit détecté (cf. section 6.3).

Pour déterminer si un modèle a été modifié, nous ne pouvons pas utiliser les changements de contexte puisque le contexte peut évoluer sans que le modèle lui-même soit modifié (e.g. demande de création alors qu'un élément existe déjà). Pour des raisons d'efficacité nous avons enrichi les actions pour mémoriser lors de l'exécution d'une transformation les appels d'actions ayant entraîné une modification du modèle, i.e. les éléments créés, les valeurs effectivement modifiées, les méta-éléments vraiment ajoutés, etc.

6.2. *Approche ensembliste*

L'approche ensembliste consiste à introduire un ensemble de politiques 'isolées' sur le même modèle. Les transformations associées aux politiques sont alors liées à un même modèle, mais les modifications résultant des applications ne sont pas accessibles. Le modèle résultant est alors calculé en quatre étapes.

Soit le système : $S = (\{\tau^i_{\sigma_i}\}, MM_{ref}, m)$ avec MM_{ref} l'ensemble des métamodèles cibles des transformations $\tau^i_{\sigma_i}$ et tel que m est le modèle résultant conforme à MM_{ref}.

Étape 1 : évaluation partielle de toutes les transformations. Chaque transformation du Système est partiellement évaluée afin de récupérer un ensemble de suites d'appels d'actions de modification résultat de $\bigcup_i eval_{partial}(\tau_i, \sigma_i))\}$.

Étape 2 : calcul de la suite d'appels d'action résultante. Etant donné l'ensemble des suites d'appels d'actions de modification obtenu nous recherchons à présent les conflits entre ces suites d'actions, comme cela a été décrit dans l'étape 2 de l'application d'une ITC (cf. §5.3).

Si des conflits sont détectés, l'évaluation de l'ensemble des ITCs échoue. Le modèle et son contexte ne sont pas modifiés. En particulier, les modifications de contexte dues à l'évaluation de la partie sélection ne sont pas mémorisées.

Étape 3 : évaluation. Si aucun conflit n'est détecté, les suites d'appels d'actions de modification sont évaluées en séquence puisque l'ordre n'a pas d'importance.

Étape 4 : itération. Si le modèle a été effectivement modifié, pour assurer la conformité vis-à-vis de tous les métamodèles, toutes les ITCs sont ré-appliquées. En effet, une action de création peut par exemple introduire un nouvel élément qu'une autre ITC devra enrichir. Nous utilisons alors la propriété d'idempotence des transformations pour ré-appliquer l'ensemble des transformations. Le point fixe est atteint lorsque le modèle n'est plus modifié. Nous nous trouvons alors dans un processus séquentiel d'applications des transformations. Le problème est d'assurer la terminaison du procédé d'applications des transformations en séquence.

6.3. *Pistes pour assurer la terminaison*

Le processus d'évolution des modèles que nous proposons repose sur des applications séquentielles et ensemblistes des transformations, ainsi que des actions de l'utilisateur. L'objectif est d'obtenir un système cohérent et de savoir le réparer dans le cas contraire. Cependant, deux problèmes demeurent : la détection de conflits de fusion interrompt le procédé et la terminaison du procédé n'est pas garantie. Nous envisageons plusieurs pistes pour pallier ces problèmes.

6.3.1. *Gestion des conflits*

Les conflits détectés sont dûs soit à des problèmes d'ordre d'évaluation entre les appels d'actions, soit à des valuations interdites.

Conflits d'ordre : Les conflits d'ordre mettent en jeu une action de suppression d'élément. Par exemple, un élément doit être détruit dans une suite d'appels d'actions alors qu'une autre demande sa création.

Une approche possible, qui est celle choisie pour gérer des cas de conflits dans des modèles distribués[MOU 10], consiste à choisir la stratégie du minimum en "oubliant" tous les appels d'actions en conflit pour ne garder que l'appel à destruction de l'élément conflictuel. Cependant cette stratégie peut garder des éléments qui n'auraient pas été créés en l'absence de cet élément et pose le problème d'une exécution seulement partielle des transformations. Une autre approche consiste à n'appliquer que les transformations qui portent les actions de destruction (si cela est possible), puis à tenter de ré-appliquer l'ensemble des transformations. Cette approche correspond à une analyse des dépendances entre transformations sur un jeu fixe d'éléments de modèles. C'est la solution que nous préférons et qui fait partie de nos perspectives, mais elle pose le problème d'un cycle potentiel présenté ci-après.

Valuations interdites : D'autres détections de conflits portent sur la valuation d'éléments et correspondent à des faux positifs : interdiction de valuation alors que la valeur de l'élément et la valeur à affecter sont les mêmes. Ce type de conflit peut être éliminé dans l'approche ensembliste puisque nous avons alors accès à toutes les transformations simultanément. Ce n'est pas le cas dans l'approche séquentielle, et donc dans le cas d'une ré-application des transformations.

6.3.2. *Détection de cycles*

Une transformation peut créer un élément qu'une autre transformation détruit. Dans une approche ensembliste un conflit est automatiquement détecté lors de l'analyse des conflits. Ce n'est pas le cas dans une approche séquentielle car l'ensemble des appels d'actions de modification n'est pas analysé. Des cycles peuvent alors se produire lors de la ré-application séquentielle des transformations. De manière générale, il y a un cycle dans l'application d'un ensemble de transformations si nous reconstruisons le même modèle après plusieurs itérations. La détection du cycle peut reposer sur la comparaison des modèles, ce qui suppose de préserver des modèles intermédiaires, ou sur les effets des transformations, ce qui réduit la complexité. Nous avons alors une présomption de cycles si en appliquant plusieurs fois les mêmes transformations contextualisées nous retrouvons les mêmes appels effectifs d'actions de modification. Par exemple un `createElement` qui se répète ou une affectation d'une même valeur suppose qu'entre temps il y a eu destruction de l'élément ou affectation d'une autre valeur. Une étude plus approfondie devrait être menée pour vérifier que le cycle est réel, mais elle suffit actuellement pour que nous déclenchions un avertissement.

6.3.3. *Existence d'un point fixe*

Le problème le plus difficile dans la ré-application des transformations, est la détection d'un modèle divergeant. Il pourrait y avoir divergence par
- (i) affectation de valeur croissante : nous n'avons par d'opérateur permettant de concaténer des suites ou des valeurs, ce type de divergence n'est donc pas possible ;
- (ii) création d'élément : la figure 16 présente un exemple de divergence par création d'élément. Ce cas ne peut se produire que si le métamodèle source d'une première transformation contient des métaéléments du métamodèle cible d'une seconde transformation et *vice versa*. Pour éviter ces divergences, il suffit donc que les métamodèles résultant de la différence entre le métamodèle cible et source de chaque transformation soient disjoints. Cette contrainte semble assez naturelle à introduire en considérant que toutes les transformations ciblent le même métamodèle source et qu'elles se distinguent par leur cible. Nous n'avons pas été jusqu'à présent confrontés dans nos applications à ce problème et n'avons pas encore introduit cette restriction d'écriture des ITCs.

Soit T1 une politique ajoutant une interface fournie à chaque composant et T2 une politique ajoutant un composant requis à chaque interface. La ré-application de T1 et T2 mène à un modèle divergent.

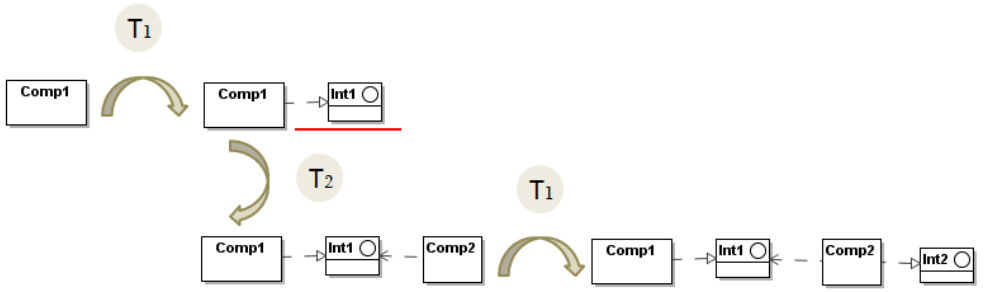
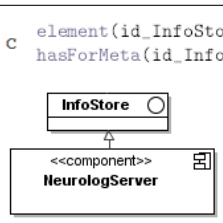


Figure 16. *Divergence par création d'un élément*

7. Mise en oeuvre, Validations et Perspectives

L'approche présentée a été implémentée en Prolog. Des transformations de Ecore vers Prolog et inversement nous permettent de manipuler les modèles de façon intégrée [MOS 09].

Basé sur la formalisation de la section 3, un modèle est exprimé par un ensemble de faits `Prolog` : `element`, `hasForMeta`, `hasForValue`. La figure 17 présente un ensemble de faits correspondants à un extrait du modèle initial (cf. figure 7). Le modèle est alors composé d'éléments (faits `element`) caractérisés par un identifiant et un nom afin de faciliter l'écriture des transformations. Les éléments sont associés à des métaéléments (faits `hasForMeta`) et peuvent être valués (faits `hasForValue`). Notons que chaque fait est créé relativement à un identifiant de contexte représenté par le dernier paramètre.




```
C  element(id_InfoStore,infoStore,init).
   hasForMeta(id_InfoStore,interface,init).

B  element(id_refNeurologServerToInfoStore,refNeurologServerToInfoStore,init).
   hasForMeta(id_refNeurologServerToInfoStore,reference,init).
   hasForValue(id_refNeurologServerToInfoStore,[id_NeurologServer,id_InfoStore],init).

A  element(id_NeurologServer,neurologServer,init).
   hasForMeta(id_NeurologServer,component,init).
```

`A` représente le composant *NeurologServer* d'identifiant *id_NeurologServer* et de nom *neurologServer* associé au métaélément *component* dans le context *init*.
`B` représente la référence entre le composant *NeurologServer* et l'interface *InfoStore*. Elle est associée au métaélément *reference* et à pour valeur la liste des identifiants des éléments qu'elle relie.
`C` représente l'interface *InfoStore* d'identifiant *id_InfoStore* et de nom *infoStore* associée au métaélément *interface* dans le context *init*.

Figure 17. *Faits `Prolog` et diagramme `UML` extrait du modèle initial*

L'écriture des ITCs est basée sur des règles `Prolog` qui implémentent les actions élémentaires. Les parties sélection, identification et modification de l'ITC correspondent à des appels en séquence à ces actions élémentaires.

Afin de de faciliter l'écriture et la compréhension des ITCs dans les modèles à composants, nous proposons une couche supplémentaire de règles. Ces *actions pour composants* permettent de sélectionner, identifier et modifier non plus des éléments, mais des composants, interfaces, paramètres d'entrée, de sortie, types et références. La figure 18 illustre l'ITC représentant partiellement la politique P_{proxy}[16].

16. La recopie des opérations et des paramètres dans les interfaces de délégation n'est pas représentée ici. L'exemple complet est disponible à cette adresse http ://modalis.polytech.unice.fr/~ clementine/PHD/policies.xhtml

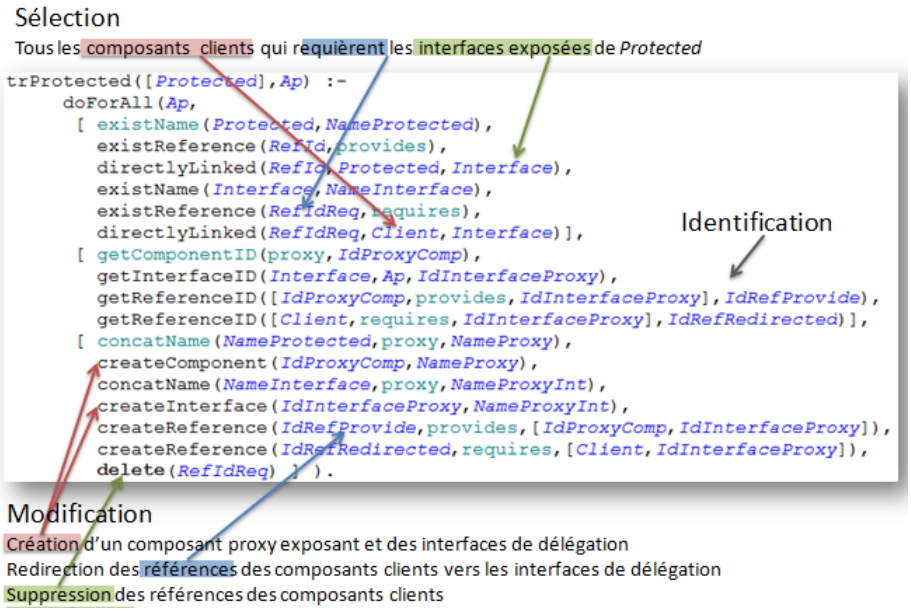


Figure 18. *Extrait de l'ITC représentant* P_{proxy}

Le maintien de la cohérence est basé sur la ré-application de l'ensemble des ITCs. Afin de mémoriser cet ensemble, chaque transformation contextualisée est enregistrée sous la forme d'un fait `Prolog`. La mise à jour du contexte se fait alors lors de l'évaluation des ITCs. Si l'évaluation est satisfaite, la base de faits est modifiée en fonction des actions de modification et de sélection. La détection de cycle est réalisée par analyse des suites d'appels d'actions de modification. Le modèle fixe est atteint lorsque l'application de l'ensemble des ITCs n'a aucun effet.

En développant les politiques de sécurité liées à notre exemple fil rouge (cf. §2.2.1) nous avons quantifié les apports de notre proposition sur un modèle comportant 300 éléments [NEM 09]. Les critères que nous avons retenus sont (i) le nombre d'éléments générés et (ii) la difficulté à trouver ses éléments à modifier dans le modèle en évaluant la distance entre les éléments générés[17] [NEM 10a]. En fonction des politiques appliquées, le nombre d'éléments générés est compris entre 20 et 50 éléments, tandis que la distance va jusqu'à 10. La ré-application des transformations a conduit à la génération de l'ordre d'une dizaine d'éléments et la distance est restée la même. Ainsi même si la réparation du modèle ne modifie pas un grand nombre d'éléments, elle aide l'utilisateur à trouver les éléments au milieu d'un modèle qu'il ne maîtrise pas à cause du nombre d'éléments ajoutés par les politiques. De plus l'automatisation de la réparation est plus sûre en évitant d'oublier des éléments.

Des transformations bien construites. Notre travail s'appuie sur les travaux précédents de validation des transformations. En particulier, nous avons comme postulats que la conformité source est préservée et que la conformité cible est atteinte par la transformation. Il n'est donc pas nécessaire d'écrire les règles de cohérence pour valider le modèle. La "justesse" des ITCs reste cependant un point difficile que des tests [BAU 06] et des approches par définition de contraintes [FOM 06] pourraient renforcer. Une fois la confiance dans les ITCs acquises alors elles jouent un rôle prépondérant de rétablissement de la conformité qu'aucun des travaux précédents ne couvre. En phase de mise au point des transformations, nous envisageons actuellement d'utiliser notre langage d'actions pour vérifier sur la base des séquences d'actions de modification calculées le respect des conformités en utilisant les travaux autour de `Praxis` [BLA 08]. Une autre perspective est la définition d'un tel système dans un langage de transformation tel que `Condor` [KNI 9] qui supporte déjà une approche logique des transformations.

De la complexité des liaisons. Nous avons défini les liaisons comme des substitutions données par l'utilisateur lors de l'application d'une politique. A l'usage nous pensons que d'autres approches devraient être utilisées comme des annotations dans les modèles ou l'utilisation de stéréotypes. Cependant, nous

17. Nous naviguons alors dans un modèle comme dans un graphe.

avons constaté qu'il peut être nécessaire d'étendre un modèle pour pouvoir lui appliquer une politique. Cette évolution n'est alors pas considérée comme faisant partie de l'application de la politique et ne peut donc ni être retirée lorsque l'on décide de "défaire" une politique, ni être réparée. L'utilisation du contexte via des "annotations dédiées" supportant une forme de traçabilité est une de nos perspectives pour pallier cette difficulté. Ainsi notre proposition offre une tracabilité automatique mais partielle de l'évolution via la gestion du contexte [AIZ 06]. Notons qu'à l'instar des travaux sur les templates [CAR 03], nous autorisons l'utilisation de type pour fixer l'expression des liaisons. Nous n'avons pas discuté ce point pour des raisons de concision.

Des modèles aux plates-formes. Nous nous sommes intéressés à des politiques exprimées sur les entités métiers et n'avons pris en charge que les conséquences au niveau des transformations vers les plates-formes. Les conséquences au niveau des couches plus basses restent donc à explorer [NAN 04]. En particulier, une des difficultés actuellement est l'utilisation simultanée de plusieurs frameworks cibles. Notre approche doit permettre de remonter les interactions entre frameworks au niveau des modèles et d'ainsi pouvoir détecter à ce niveau les conflits potentiels, voire proposer des solutions dédiées en fonction des compositions ciblées [BOU 05].

8. Conclusion

La conception des systèmes d'informations requiert la prise en compte de propriétés multiples non fonctionnelles que nous nommons politiques. La complexité pour introduire ces politiques est alors prise en charge par des transformations dédiées qui visent à réduire la tâche du concepteur, tout en assurant la cohérence des modèles par une approche systématique.
L'évolution constante des SI introduit des facteurs de complexité supplémentaires. Les politiques sont le plus souvent introduites séquentiellement au fil de l'histoire du SI. Un calcul a priori d'un ordre dans l'application des transformations n'est pas toujours possible. Les modifications indispensables du SI nécessitent de s'assurer que les propriétés portées par les politiques sont bien vérifiées. Un des défis est donc de savoir réparer les modèles ainsi construits. Pour répondre à cette problématique, nous avons montré que les politiques peuvent être définies comme des transformations idempotentes qui permettent alors de réparer automatiquement les modèles. Nous avons ainsi défendu une approche de construction qui repose sur une évolution incrémentale des modèles et une réparation ensembliste. La démarche proposée a été étayée par une implémentation en Prolog du système et a été appliquée à un système existant dans le cadre d'un projet ANR.

9. Bibliographie

[AIZ 06] AIZENBUD-RESHEF N., NOLAN B. T., RUBIN J., SHAHAM-GAFNI Y., « Model traceability », *IBM Systems Journal*, vol. 45, n° 3, 2006, p. 515-526, Citeseer.

[BAN 06] BANIASSAD E., CLEMENTS P. C., ARAUJO J., MOREIRA A., RASHID A., TEKINERDOGAN B., « Discovering Early Aspects. », *IEEE Software*, vol. 23, n° 1, 2006, p. 61-70, IEEE Computer Society Press.

[BAR 06] BARAIS O., LAWALL J., MEUR A.-F. L., DUCHIEN L., « Safe Integration of New Concerns in a Software Architecture », *13th Annual IEEE International Conference on Engineering of Computer Based Systems ECBS'06*, Potsdam, Germany, mar 2006.

[BAR 08] BARAIS O., KLEIN J., BAUDRY B., JACKSON A., CLARKE S., « Composing Multi-view Aspect Models », *Seventh International Conference on CompositionBased Software Systems ICCBSS 2008*, , 2008, p. 43–52, Ieee.

[BAT 04] BATORY D., SARVELA J. N., RAUSCHMAYER A., « Scaling step-wise refinement », *IEEE Transactions on Software Engineering*, vol. 30, n° 6, 2004, p. 355-371, IEEE Computer Society.

[BAU 06] BAUDRY B., DINH-TRONG T., MOTTU J.-M., SIMMONDS D., FRANCE R., GHOST S., FLEUREY F., LE TRAON Y., « Model Transformation Testing Challenges », *ECMDA workshop on Integration of Model Driven Development and Model Driven Testing.(IMDT 06)*, juillet 2006.

[BEI 07] BEISIEGEL M., BOOZ D., « SCA Policy Framework », rapport, mars 2007, OSOA.

[BÉZ 05] BÉZIVIN J., JOUAULT F., « Using ATL for Checking Models », *ternational Workshop on Graph and Model Transformation(GraMoT)*, septembre 2005.

[BLA 06] BLAY-FORNARINO M., FRANCHI P., « *Les langages et l'IDM* », chapitre 5, p. 99-128, 2006.

[BLA 08] BLANC X., MOUNIER I., MOUGENOT A., MENS T., « Detecting model inconsistency through operation-based model construction », *ICSE '08 : Proceedings of the 30th international conference on Software engineering*, New York, NY, USA, 2008, ACM, p. 511–520.

[BLA 09] BLANC X., MOUGENOT A., MOUNIER I., MENS T., « Incremental Detection of Model Inconsistencies Based on Model Operations », *CAiSE*, 2009, p. 32-46.

[BOU 05] BOUZITOUNA S., GERVAIS M.-P., BLANC X., « Model Reuse in MDA. », ARABNIA H. R., REZA H., Eds., *Software Engineering Research and Practice*, CSREA Press, 2005, p. 354–360.

[BRO 06] BROTTIER E., FLEUREY F., STEEL J., BAUDRY B., TRAON Y., « *Metamodel-Based test generation for model transformations : an algorithm and a tool* », vol. 1998, p. 85-94, IEEE, 2006.

[CAR 03] CARON O., CARRÉ B., MULLER A., VANWORMHOUDT G., « A Framework for Supporting Views in Component Oriented Information Systems », *in OOIS 2003*, vol. 2817 de *Lecture Notes in Computer Science*, Geneva, Switzerland, September 2003, Springer Verlag, p. 164–178.

[CAS 08] CASEAU Y., *Urbanisation, SOA et BPM - Le point de vue d'un DSI*, Dunod édition, 2008.

[CHA 04] CHARFI A., MEZINI M., « Aspect-Oriented Web Service Composition with AO4BPEL. », *ECOWS*, vol. 3250 de *LNCS*, Springer, 2004, p. 168-182.

[CHA 08] CHADWICK D. W., SU L., LABORDE R., « Coordinating access control in grid services », *Concurrency and Computation : Practice and Experience*, vol. 20, n° 9, 2008, p. 1071-1094.

[CLA 01] CLARKE S., WALKER R. J., « *Composition patterns : an approach to designing reusable* », p. 5-14, IEEE Comput. Soc, 2001.

[CLA 02] CLARKE S., « Extending standard UML with model composition semantics », *Sci. Comput. Program.*, vol. 44, n° 1, 2002, p. 71–100, Elsevier North-Holland, Inc.

[CRÉ 10] CRÉGUT X., COMBEMALE B., PANTEL M., FAUDOUX R., PAVEI J., « Generative technologies for model animation in the TopCased platform », *6th European Conference on Modelling Foundations and Applications(ECMFA 2010)*, vol. 6138 de *Lecture Notes in Com*, Paris, France, juin 2010, Springer, p. 90–103.

[ENG 02] ENGELS G., HECKEL R., KÜSTER J. M., GROENEWEGEN L., « Consistency-preserving model evolution through transformations », *2002 The Unified Modeling Language*, , 2002, p. 212-227, Springer.

[FOM 06] DE FOMBELLE G., BLANC X., RIOUX L., GERVAIS M.-P., « Finding a Path to Model Consistency », *European Conference on Model Driven Architecture - Foundations and Applications(ECMDA-FA 2006)*, vol. 4066 de *LNCS*, juillet 2006.

[FOM 07] DE FOMBELLE G., « Gestion incrémentale des propriétés de cohérence structurelle dans l'ingénierie dirigée par les modèles », PhD thesis, université Pierre et Marie Curie, Paris VI, septembre 2007.

[GAI 09] GAIGNARD A., MONTAGNAT J., « A distributed security policy for neuroradiological data sharing », *HealthGrid'09*, IOS Press, juin 2009, p. 257–262.

[GAM 93] GAMMA E., HELM R., JOHNSON R., VLISSIDES J., *Design Patterns : Abstraction and Reuse of Object-Oriented Design*, vol. 707, 1993.

[GER 02] GERVAIS M.-P., « Towards an MDA-oriented methodology », *26th annual international computer software and applications conference(COMPSAC 02)*, Oxford, août 2002, p. 265-270.

[GIR 06] GIRAUDIN J.-P., « Complexité des systèmes d'information et de leur ingénierie », *Revue électronique internationale e-TI (e-revue en Technologies de l'Information), Numéro 3*, , 2006.

[GUE 00] GUENNEC A. L., SUNYÉ G., JÉZÉQUEL J.-M., « Precise Modeling of Design Patterns », EVANS A., KENT S., SELIC B., Eds., *UML 2000 - The Unified Modeling Language. Advancing the Standard. Third International Conference, York, UK, October 2000, Proceedings*, vol. 1939, Springer, 2000, p. 482–496.

[HA 02] HA S.-C., CHOI J.-E., « A Model Design of the Track and Trace System for e-Logistic », *Operationel Research*, vol. 2, n° 1, 2002, p. 5-15.

[KIC 97] KICZALES G., LAMPING J., MENDHEKAR A., MAEDA C., LOPES C., LOINGTIER J.-M., IRWIN J., « Aspect-Oriented Programming », *Proceedings of the European Conference on Object-Oriented Programming*, vol. 1241, 1997, p. 220–242.

[KNI 9] KNIESEL G., « Detection and Resolution of Weaving Interactions », *Transactions on Aspect-Oriented Programming, special issue on Aspect Dependencies and Interactions (edited by R. Chitchyan, J. Fabry, S. Katz, A. Rensink)*, vol. LNCS, n° April 2009, 2006-9, p. 1-53, Springer.

[LAJ 10] LAJMI A., CAUVIN S., ZIANE M., ZIADI T., « A Multi-View Model-Driven Approach for Packaging Software Components », *25th Annual Symposium on Applied Computing(SAC 2010)*, ACM, mars 2010.

[LOD 02] LODDERSTEDT T., BASIN D., DOSER J., « *SecureUML : A UML-Based Modeling Language for Model-Driven Security* », vol. 2460, p. 426-441, ACM Press, 2002.

[MEN 02] MENS K., MICHIELS I., WUYTS R., « Supporting Software Development Through Declaratively Codified Programming Patterns », *journal on Expert Systems with Applications*, vol. 23, n° 4, 2002, p. 405-413, Elsevier Science.

[MEN 06] MENS T., KNIESEL G., RUNGE O., « Transformation dependency analysis. A comparison of two approaches », *Langages et Modèles à Objets(LMO)*, Hermes, mars 2006, p. 167-182.

[MOS 09] MOSSER S., BLAY-FORNARINO M., « Réflexions autour de la construction dirigée par les modèles d'un atelier de composition d'orchestrations », *15ème conférence francophone sur les Langages et Modèles à Objets(LMO'09)*, Cépadues, mars 2009.

[MOU 10] MOUGENOT A., « Praxis : Detection of inconsistency within distributed models », PhD thesis, UPMC, 2010.

[MUL 05] MULLER A., CARON O., CARRE B., VANWORMHOUDT G., « On Some Properties of Parameterized Model Application », *First European Conference on Model Driven Architecture - Foundations and Applications(ECMDA-FA'2005)*, A. Hartman and D. Kreische Eds., novembre 2005.

[NAN 04] NANO O., BLAY-FORNARINO M., « Annotations et transformations de modèles pour l'intégration de services », *RSTI - Série L'Objet (RSTI-Objet)*, vol. 10, n° 2-3, 2004, p. 175-188.

[NEM 09] NEMO C., BLAY-FORNARINO M., « Collaborative Modeling for Integrating Security Properties on a Medical Sharing Application », rapport, juillet 2009, University of Nice, I3S CNRS, Sophia Antipolis, France.

[NEM 10a] NEMO C., « Construction et validation de modèles guidées par l'application idempotente de transformations », PhD thesis, Nice - Sophia Antipolis Universit, Sophia Antipolis (France), décembre 2010.

[NEM 10b] NEMO C., BLAY-FORNARINO M., « Construction of Models Needs Idempotent Transformations », *Sophia-Antipolis Formal Analysis Group, workshop*, , Sophia Antipolis, France, octobre 2010, INRIA, p. 1-4.

[OMG 10] OMG C., « Object Constraint Language (Version 2.2) », février 2010.

[PAW 05] PAWLAK R., DUCHIEN L., SEINTURIER L., « CompAr : Ensuring Safe Around Advice Composition », STEFFEN M., ZAVATTARO G., Eds., *Formal Methods for Open Object-Based Distributed Systems, 7th IFIP WG 6.1 International Conference, FMOODS 2005, Athens, Greece, June 15-17, 2005, Proceedings*, vol. 3535 de *Lecture Notes in Computer Science*, Springer, 2005, p. 163–178.

[POU 07] POURRAZ F., « Diapason : une approche formelle et centrée architecture pour la composition évolutive de services Web », PhD thesis, Université de Savoie, LISTIC, december 2007.

[SAD 97] SADOHARA K., HARAGUCHI M., « Using Abstraction Schemata in Inductive Logic Programming », *ILP '97 : Proceedings of the 7th International Workshop on Inductive Logic Programming*, London, UK, 1997, Springer-Verlag, p. 256–263.

[SIL 10] DA SILVA M. A. A., MOUGENOT A., BLANC X., BENDRAOU R., « Towards Automated Inconsistency Handling in Design Models », PERNICI B., Ed., *CAiSE*, vol. 6051 de *Lecture Notes in Computer Science*, Springer, 2010, p. 348-362.

[SIM 05] SIMMONDS D., SOLBERG A., REDDY R., FRANCE R., GHOST S., « An Aspect Oriented Model Driven Framework », *International Enterprise Distributed Object Computing Conference(EDOC 2005)*, septembre 2005.

[STE 04] STEEL J., JÉZÉQUEL J.-M., « Typing Relationships in MDA », *Second European Workshop on Model Driven Architecture(EWMDA 2)*, D. H. Akehurst, 2004.

[STE 10] STEVENS P., « *Bidirectional Model Transformations in QVT : Semantic Issues and Open Questions* », janvier 2010.

[STR 04] STRAW G., GEORG G., SONG E., GHOSH S., FRANCE R., BIEMAN J. M., « Model Composition Directives », *in Proceedings of UML'2004 :7th International Conference on UML Modeling Languages and Applications*, vol. 3273 de *Lecture Notes in Computer Science*, Lisbon, Portugal, October 2004, p. 84–97.

[VAN 03] VAN GORP P., STENTEN H., MENS T., DEMEYER S., « Towards Automating Source-Consistent UML Refactorings », , 2003, p. 144-158, Springer Verlag.

[VIR 04] VIROLI M., « Towards a Formal Foundation to Orchestration Languages. », *Electr. Notes Theor. Comput. Sci.*, vol. 105, 2004, p. 51-71.

[W3C 07] W3C, « Web Services Policy 1.5 - Framework », rapport, 2007.

[WAG 08] WAGELAAR D., « Composition techniques for rule-based model transformation languages », *Theory and Practice of Model Transformations*, , 2008, p. 152-167, Springer.