ANR programme ARPEGE 2008

Systèmes Embarqués et Grandes Infrastructures

Projet SELKIS : Une méthode de développement de systèmes d'information médicaux sécurisés : de l'analyse des besoins à l'implémentation.

ANR-08-SEGI-018

Février 2009 - Décembre 2011

Modelling and verification of the Res@mu model

Livrable numero 6.1.2

Akram Idani, Yves Ledru, Jean-Luc Richier, Mohamed-Amine Labiadh
Laboratoire d'Informatique de Grenoble

Septembre 2011

Table des matières

Introduction

Ce livrable présente en partie les résultats de l'application de nos contributions théoriques et pratiques sur l'étude de cas Res@mu. En particulier, il s'attache à l'application des principes de la traduction des modèles graphiques UML en B. Notons que ces principes ont été énoncés dans le livrable 3.1 (chapitre 4) et mis à jour dans le livrable 3.2 (chapitre 6). Lesdits livrables (3.1 et 3.2) s'inscrivent dans le cadre des travaux entrepris par le LIG autour du couplage de modèles graphiques et formels ainsi que ceux proposés par le LACL autour des ASTD (Algebraic State Transition Diagrams). Ces travaux sont motivés par l'usage d'outils de preuve et d'animation assistant les méthodes formelles telle que la méthode B en vue d'effectuer des vérifications automatisées de modèles graphiques.

Nous ne présenterons pas toutes les traductions B issues des divers modèles graphiques de l'étude de cas Res@mu, mais plutôt la démarche adoptée ainsi que les résultats que nous avons obtenus lors de la traduction du modèle fonctionnel. La formalisation graphique des règles de sécurité et les spécifications B correspondantes feront l'objet d'une mise à jour future du présent livrable.

Rappelons qu'une première analyse des besoins pour Res@mu a été réalisée dans le livrable 6.1.1 où une première ébauche du modèle fonctionnel a été présentée par les collègues de IFREMMONT. Nous reprenons ici les principes élémentaires de ce modèle en mettant l'accent sur les compléments qui nous ont été fournis en vue d'affiner ce modèle et le rendre traduisible en un langage formel.

Ce livrable suivra l'enchaînement suivant :

- Le chapitre 1 donne une vue d'ensemble de l'étude de cas Res@mu et récapitule les concepts du modèle fonctionnel sur lesquels nous nous sommes basés dans notre étude.
- Le chapitre 2 fait le point sur l'automatisation du processus de transformation en mettant l'accent sur les paradigmes que nous avons mis en pratique pour générer les modèles B.
- Le chapitre 3 présente une vue d'ensemble des spécifications B obtenues à partir du modèle fonctionnel ainsi que la démarche entreprise pour leur validation.

Chapitre 1

Analyse du modèle Res@mu

1.1 Introduction

Le modèle Res@mu sur lequel nous avons appliqué nos contributions se présente sous la forme d'un ensemble de modèles UML élaborés au moyen de l'outil StarUML[1]. Bien que ces modèles soient exhaustifs, car ils abordent de nombreuses exigences fonctionnelles, ils restent dédiés à une activité d'analyse. En effet, ils mettent en jeu principalement des diagrammes de cas d'utilisation et des diagrammes de classes de haut niveau. Néanmoins, les spécifications formelles que nous cherchons à produire à partir du modèle Res@mu se doivent d'être suffisamment précises en vue de pouvoir réaliser les vérifications associées. C'est pourquoi, nous avons été amenés à étudier plus finement ce modèle dans le but d'en élaborer un modèle de conception impliquant des élément structurels de plus bas niveau (attributs, types d'attributs, multiplicités, rôles, etc).

Pour ce faire, nous avons analysé, dans un premier temps, les cas d'utilisation afin d'identifier ceux qui impliquent des entités fonctionnelles potentiellement critiques, tels que les actes de soins, les enregistrements liés à des patients, la constitution des équipes d'intervenants... Nous avons ensuite sollicité des scénarios détaillés relatifs aux cas d'utilisation choisis. L'analyse de ces scénarios a débouché sur un diagramme de classes logicielles suffisamment précis et pouvant être traité par notre plateforme de génération de spécifications B. Notons qu'il s'agit ici uniquement du modèle fonctionnel. Les exigences de sécurité sont informelles et ont été dégagées progressivement selon la démarche décrite dans le livrable 6.1.1.

Ce chapitre donne donc une vue d'ensemble du modèle Res@mu et met l'accent sur les différents choix qui ont été faits pour pouvoir le traduire en B et entamer notre démarche de validation.

1.2 Vue d'ensemble du modèle Res@mu

Globalement, le modèle Res@mu tel qu'il a été conçu avec StarUML se compose de :
- 15 paquetages,
- 77 classes (avec des arités incomplètes),
- 100 use cases,
- 0 acteur

La figure 1.1 donne un aperçu de l'explorateur du modèle tel qu'il se présente sous StarUML. Remarquons que chaque paquetage est constitué d'un diagramme de cas d'utilisation et d'un diagramme de classes.

[1] http ://staruml.sourceforge.net

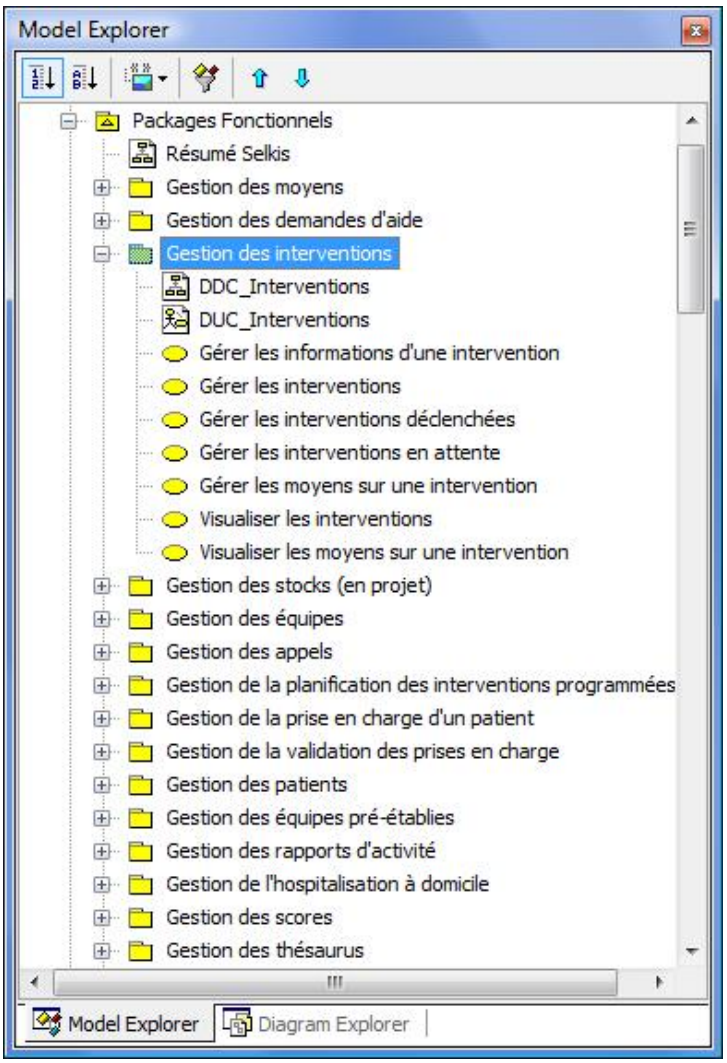

FIG. 1.1 – Extrait de l'explorateur StarUML du modèle Res@mu

Cette organisation sous forme de paquetages a permis de faciliter notre démarche d'identification du sous-ensemble nécessaire à l'usage de notre outil de génération de spécifications B.

1.2.1 Analyse des diagrammes de cas d'utilisation

Lors de cette analyse nous nous sommes intéressés aux cas d'utilisation mettant en jeu les actes de soins et les patients. Notre intérêt a particulièrement porté sur les paquetages : gestion des interventions, gestion des équipes et gestion des patients avec un focus sur les classes *ManagementAct* et *Patient*. De ce fait, le nombre de cas d'utilisation potentiellement intéressants est réduit à 28 cas d'utilisation parmi les 100 cas d'utilisation du modèle Res@mu.

Malgré ce choix d'un sous-ensemble du modèle, diverses questions se sont naturellement posées dans le but d'aboutir à un modèle de conception plus fin et plus précis. Par exemple, les cas d'utilisation de la figure 1.2, relatifs à des actes de soin réalisés sur place ou à distance, permettent de dégager plusieurs questions intéressantes du point de vue fonctionnel :

– Que signifie l'héritage entre cas d'utilisation ? Est-ce que cela implique des spécialisations de la classe *ManagementAct* selon que le patient soit pris en charge localement ou à distance ? Ou est-ce qu'il implique des procédures similaires entre “Manage a patient locally” et “Manage a

patient remotely" ?
– Le cas d'utilisation "Manage a patient" est-il abstrait ? si c'est le cas alors aucune autre procédure de prise en charge ne peut être envisagée !
– La validation est-elle obligatoire ou optionnelle ?

Ces exemples de questions sont issues des ambiguïtés des notations graphiques (e.g. héritage/extension) qui méritent d'être levées avant d'entamer notre étape de traduction en B. Dans la figure 1.2 on remarque que la validation est modélisée par une extension, ce qui peut implicitement faire référence au caractère optionnel de ce cas d'utilisation. Toutefois, en affinant ce point nous avons remarqué que la validation est obligatoire dans les deux cas, qu'il s'agisse d'une prise en charge sur place ou d'une prise en charge à distance.

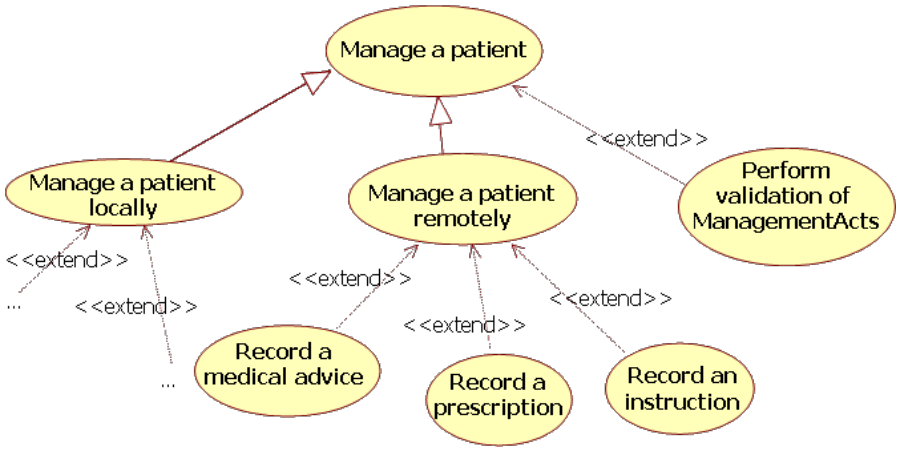

FIG. 1.2 – Sous-ensemble de cas d'utilisation relatifs à la prise en charge d'un patient

Notons par ailleurs, que nous ne nous sommes pas attachés à la répartition d'acteurs en fonction des cas d'utilisation étant donné que nous avons proposé, dans le livrable 6.1.1, une reformulation des divers cas d'utilisation sous forme de buts KAOS. Ces derniers sont accessibles par des agents dont les relations ont été explicitées par une hiérarchie d'agents. Nous reviendrons plus amplement sur cette notion d'agent dans une future mise à jour du présent livrable quand nous aborderons les diverses règles de contrôles d'accès que nous avons élaborées pour le modèle Res@mu.

1.2.2 Analyse du diagramme de classes

Le diagramme de classes d'analyse est constitué dans son ensemble de 77 classes. Leur traduction directe en B pourrait produire des spécifications de grande taille et difficiles à aborder. D'une part, parce que de nombreuses classes sont issues de cas d'utilisation autres que ceux choisis dans la section précédente, et d'autre part, parce que les classes d'analyse sont exemptes de détails nécessaires au processus de génération de spécifications B (*e.g.* types d'attributes, multiplicités, rôles, ...).

La figure 1.3 présente un sous-ensemble du diagramme de classes issu du paquetage "Gestion des interventions". Ce diagramme montre que les deux entités *ManagementAct* et *Patient* [2] sur lesquelles nous nous basons dans ce travail sont fortement connectées (liens d'héritage, de composition et associations simples) à d'autres classes du domaine. Un effort supplémentaire

[2]Les classes *ManagementAct* et *Patient* sont entourées en noir.

a donc été fait en vue de pouvoir trouver les liens les plus pertinents et ce en exploitant des scénarios d'exécution concrets.

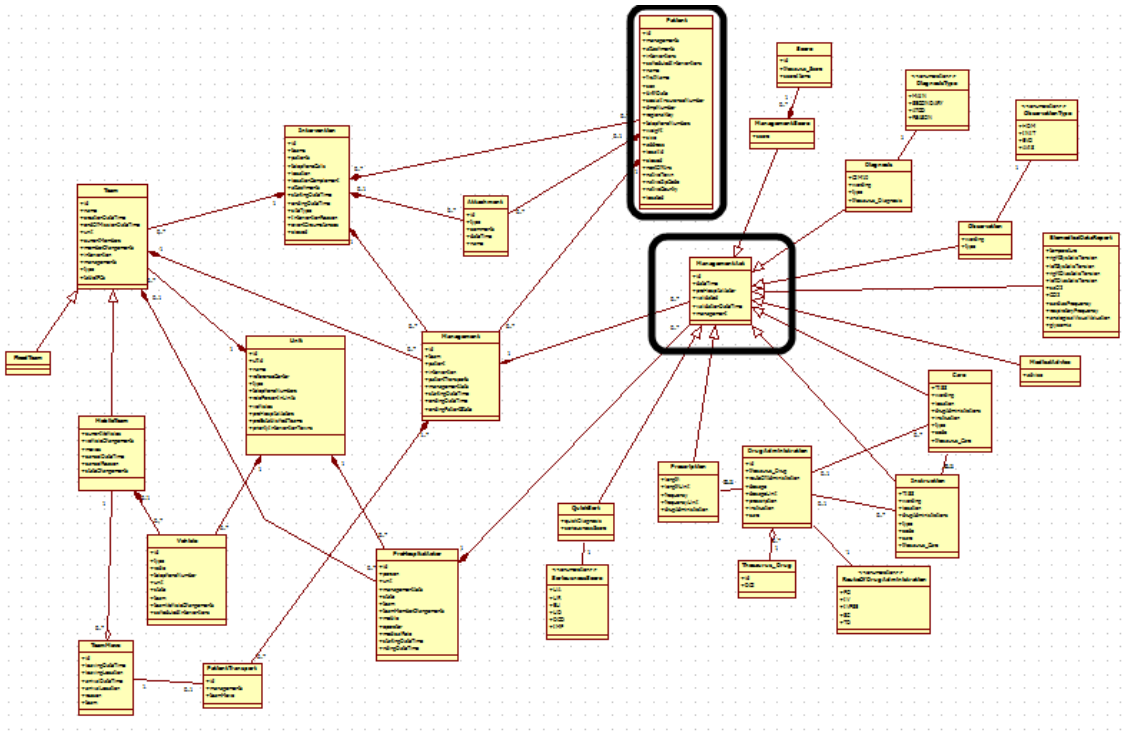

FIG. 1.3 – Extrait d'un sous-ensemble du diagramme de classes Res@mu

1.3 Scénarios d'exécution

Les scénarios que nous avons sollicités en vue d'aboutir à un diagramme de classes logicielles traduisibles en B sont proches de l'exécution réelle de l'application. Ils permettent donc de montrer concrètement les interactions entre les diverses entités du système d'information. Nous avons ainsi étudié cinq scénarios majeurs relatifs au processus de prise en charge d'un patient :

– démarrer prise en charge,
– valider prise en charge,
– invalider prise en charge,
– enregistrer prise en charge,
– terminer prise en charge.

Ces diagrammes de séquences font référence à des classes de bas niveau spécifiques à la plateforme Hibernate. Celle-ci est utilisée par IFREMMONT pour mettre en place le système d'information. Parmi ces classes, nous identifions les classes *Facade* (e.g. PatientFacade) qui constituent le point d'entrée de l'utilisateur au système, les classes *Manager* (e.g. PatientManager) qui jouent le rôle de contrôleurs et *HibernateTemplate* qui correspond à la couche persistance. Dans la suite nous n'allons pas étudier ces classes car elles ne sont pas nécessaire dans le modèle PIM (indépendant de la plateforme), et ne contribuent pas à nos objectifs de validation.

Démarrer prise en charge

La figure 1.4 correspond au démarrage d'une prise en charge d'un patient. Pour ce faire, un patient et une équipe doivent avoir été créés au préalable. Ceci est justifié par les opérations *getPatient* et *getTeam*. Le processus de création des instances des classes Patient et Team précède celui du démarrage d'une prise en charge. En effet, lors d'un appel téléphonique signalant une situation d'urgence, une instance d'une classe intervention est créée à laquelle sont associés une équipe (choisie parmi des équipes pré-établies) et le patient sujet de l'intervention.

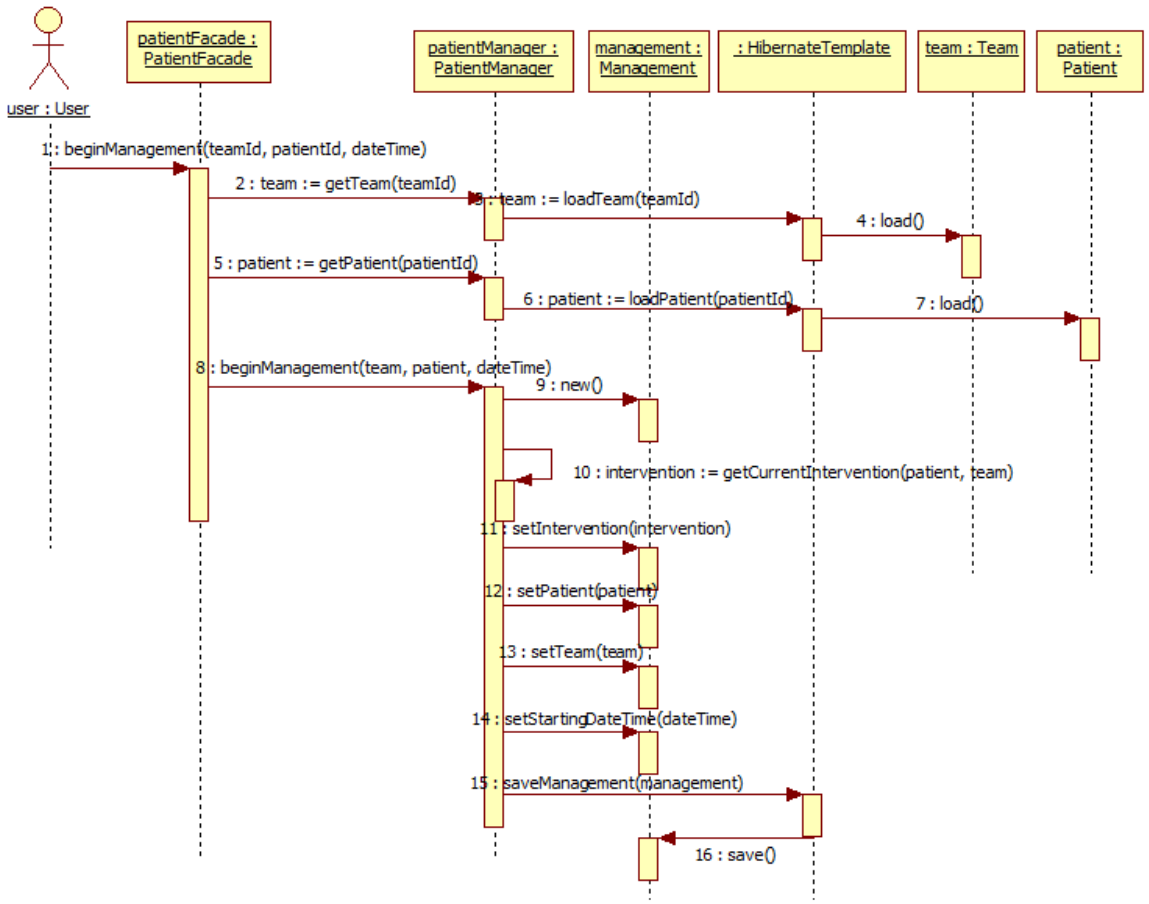

FIG. 1.4 – Scénario d'exécution : démarrer prise en charge

Le démarrage d'une prise en charge se traduit donc par la création d'une nouvelle instance de la classe *Management* à laquelle sont associés l'intervention, le patient, l'équipe affectée à l'intervention ainsi qu'une date de démarrage.

Terminer prise en charge

La fin d'une prise en charge est présentée par le diagramme de séquences de la figure 1.5. Celui-ci met à jour deux informations : la date de fin de la prise en charge (attribut *endingDateTime* de la classe *Management*) et l'état du patient (attribut *endingPatientState* de la classe *Management*).

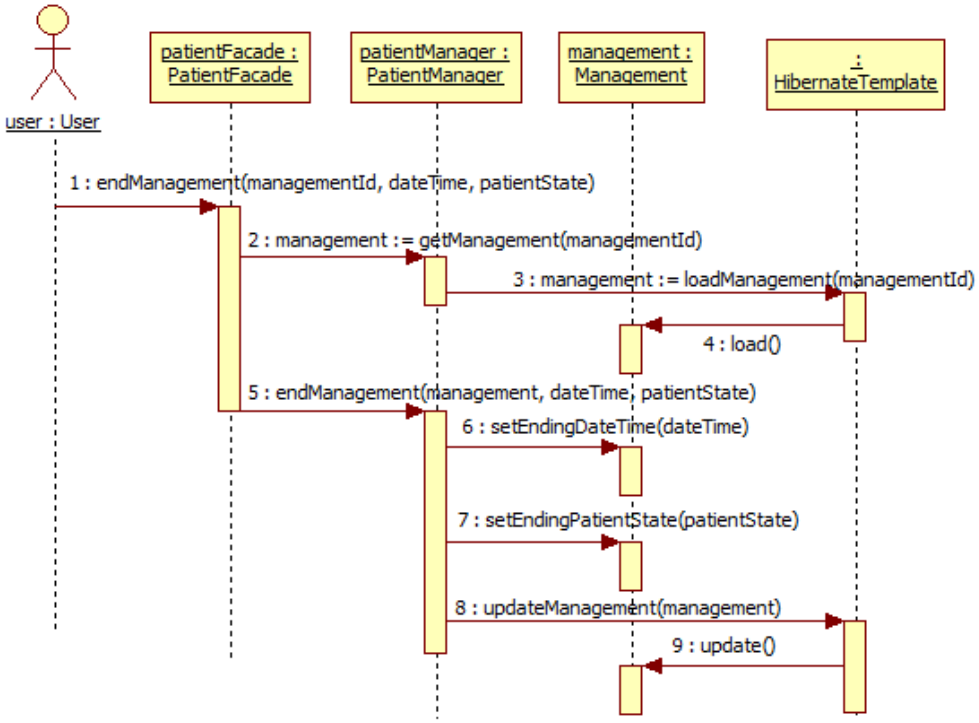

FIG. 1.5 – Scénario d'exécution : terminer prise en charge

Enregistrer prise en charge

L'enregistrement d'une prise en charge (figure 1.6) fait référence à la réalisation d'un acte de soin par un acteur pré-hospitalier (instance de la classe *PreHospitalActor*).

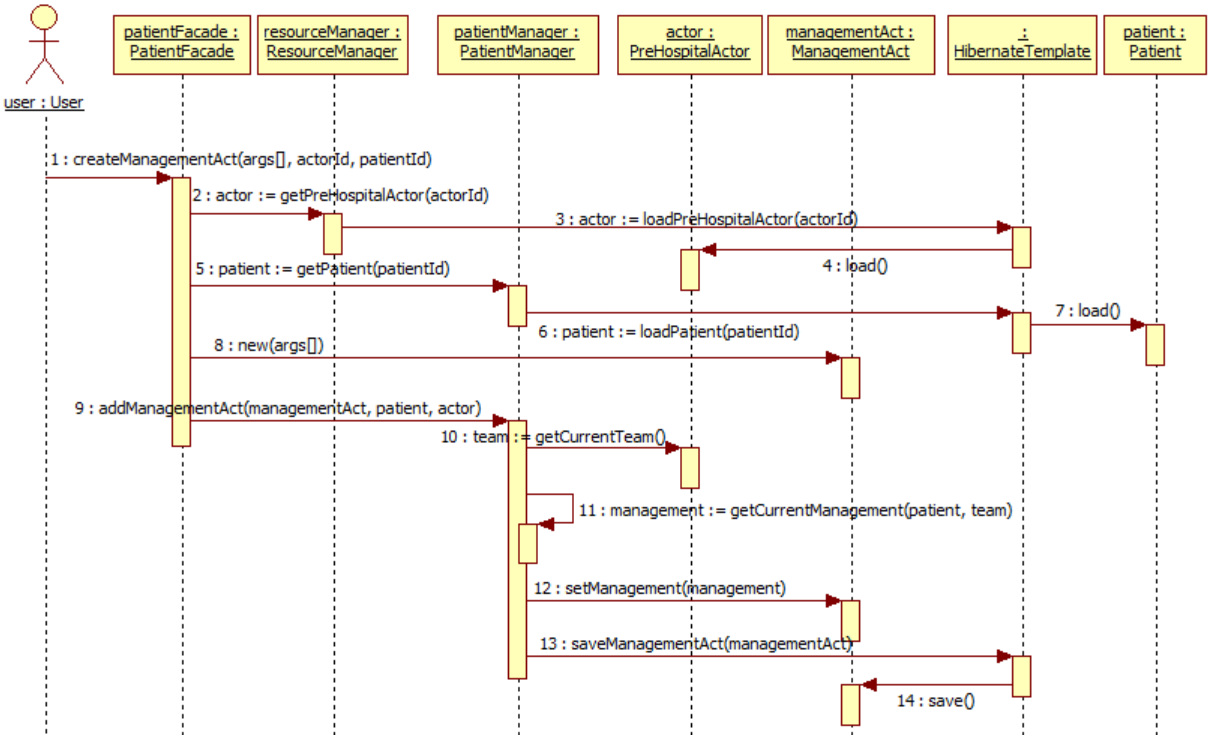

FIG. 1.6 – Scénario d'exécution : enregistrer prise en charge

Pour ce faire, le système doit d'abord connaître d'un côté l'acteur pré-hospitalier à l'origine de l'acte ainsi que le patient. Ceci se traduit par les opérations *GetPreHospitalActor* et *GetPatient*. Cela a pour but de retrouver l'équipe chargée du patient et de vérifier par conséquent que l'acteur pré-hospitalier est en mesure de réaliser l'acte de soin de par son appartenance à cette équipe. Ensuite, le système crée une nouvelle instance de la classe *ManagementAct* indiquant ainsi qu'un acte de soin a été réalisé et que les informations associées peuvent être enregistrées. Cette instance de *ManagementAct* sera par la suite liée à l'instance de la classe *Management* relative à la prise en charge en cours ainsi qu'à l'acteur pré-hospitalier qui l'a initié.

Valider prise en charge

La validation d'une prise en charge présentée au niveau de la figure 1.7 met en jeu, non pas la classe *Management*, ni la classe *Intervention*, mais plutôt la classe *ManagementAct*. Notons que dans le diagramme de classes, une prise en charge (instance de la classe *Management*) peut être composée de plusieurs actes de soins (instances de la classe *ManagementAct*). Ainsi, la validation d'une prise en charge peut se traduire par la validation de tous les actes de soin associés à cette prise en charge. Concrètement, le déroulement de ce scénario fait appel à l'opération *setValidated* qui met à vrai l'attribut booléen *validated* de la classe *ManagementAct*.

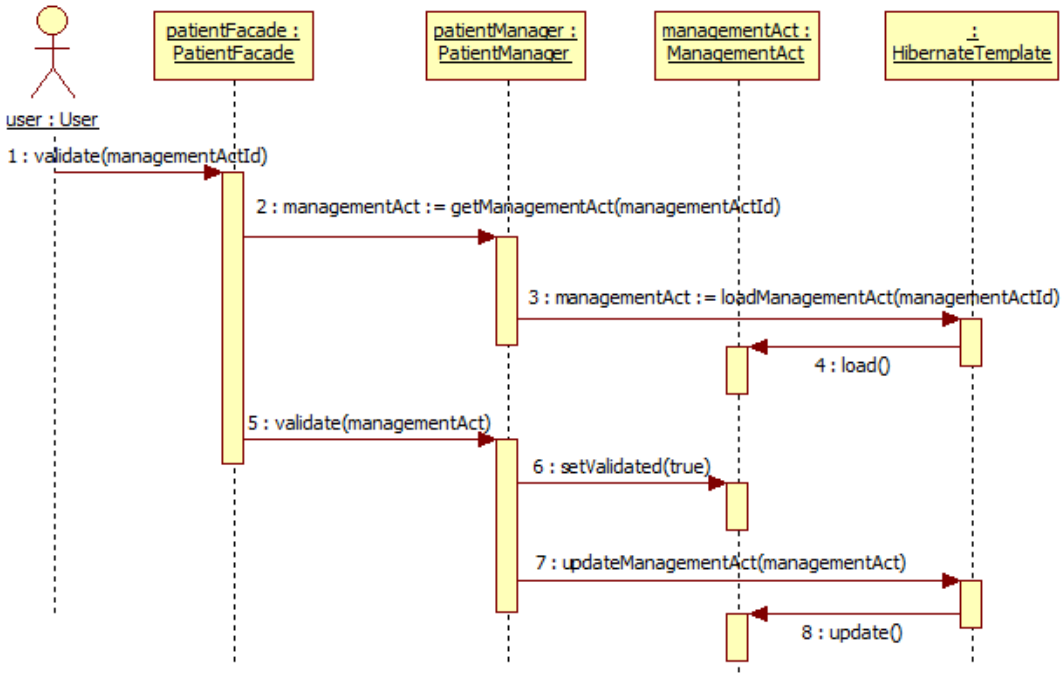

Fig. 1.7 – Scénario d'exécution : valider prise en charge

Invalider prise en charge

La non validation d'un acte de soin est présentée par la figure 1.8.

Ce scénario nécessite, d'une part l'indication d'une raison de non validation, et d'autre part l'enregistrement de la date de cette opération. Il fait ainsi référence aux attributs *invalidationReason* et *validationDateTime*.

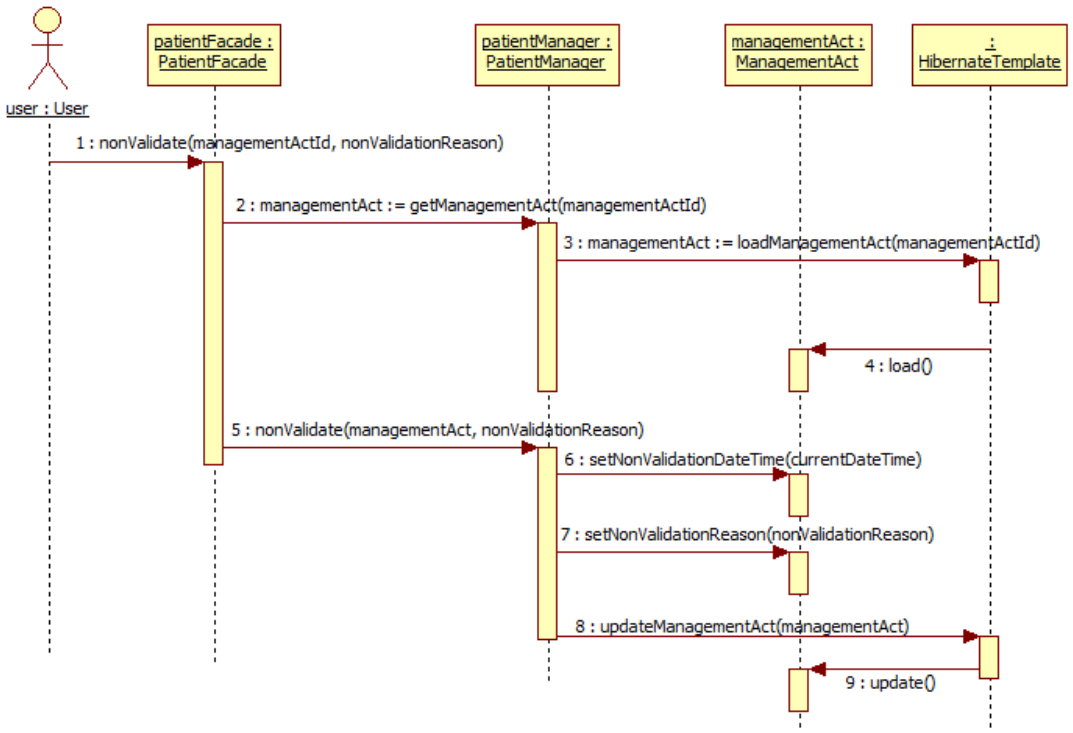

FIG. 1.8 – Scénario d'exécution : invalider prise en charge

1.4 Diagramme de classes final

Les diagrammes de séquences présentés dans la section précédente nous ont permis d'avoir une idée plus précise du déroulement des prises en charge de patients. Ils mettent en valeur les classes, les attributs de classes ainsi que les opérations requises. Sur cette base, nous avons pu délimiter et enrichir un sous-ensemble du diagramme de classes d'analyse. Le diagramme de classes obtenu est présenté par la figure 1.9.

Les acteurs pré-hospitaliers

Ces acteurs sont représentés par les classes *Person* et *PreHospitalActor*. Les rôles qu'ils peuvent avoir (Doctor, Nurse, Parm, Rescuer...) sont modélisés par l'attribut *medicalRole* de la classe *PreHospitalActor*. L'attribut *state* est nécessaire lors de la constitution des équipes et lors de la modification d'une équipe en cours d'intervention. Il indique si l'acteur est disponible, s'il est en cours d'intervention ou s'il est indisponible.

Constitution des équipes

La classe *PreEstablishedTeam* désigne des équipes pré-établies composées chacune d'un ou de plusieurs acteurs pré-hospitaliers. L'existence de ces équipes permet de lancer rapidement une intervention en évitant de se focaliser sur la constitution d'une nouvelle équipe en situation d'urgence. En effet, l'opérateur, qui déclenche une intervention, choisit une équipe parmi celles qui sont pré-établies et constitue ainsi l'instance de la classe *Team* chargée du patient.

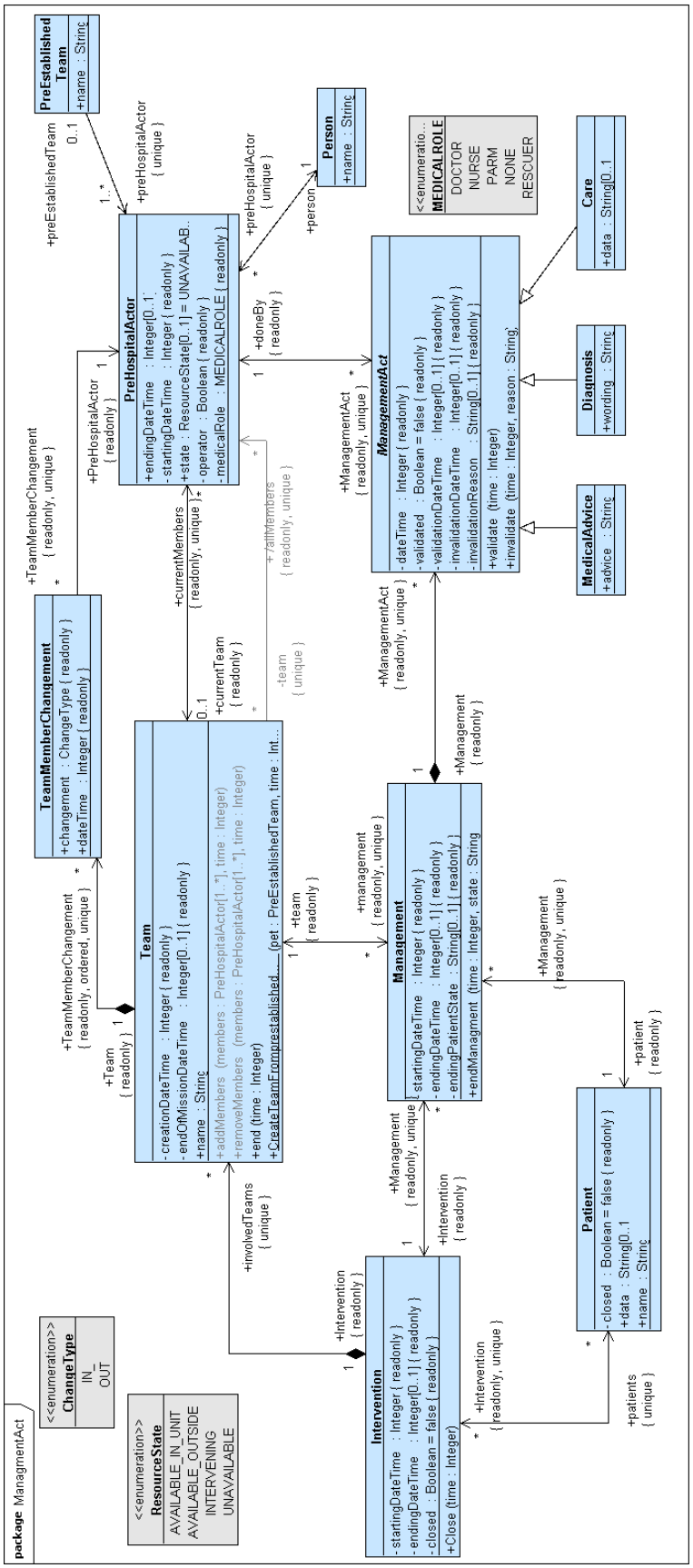

FIG. 1.9 – Diagramme de classes choisi

Cependant, durant une intervention une équipe (instance de la classe *Team*) est susceptible de changer de par l'arrivée ou le départ d'acteurs pré-hospitaliers. Le lien entre les classes *Team* et *PreHospitalActor* représente les membres courants d'une équipe d'intervention. La classe *TeamMemberChangement* permet de retracer les évolutions de ce lien en enregistrant le moment du changement (attribut *dateTime*), l'acteur pré-hospitalier concerné par ce changement ainsi que le type du changement (arrivée ou départ). Cette trace est nécessaire car l'accès aux informations d'un patient est conditionné par l'appartenance à l'équipe qui s'est chargée de ce patient.

Les interventions

Les interventions sont représentées par la classe *Intervention*. À une intervention peuvent être associés plusieurs équipes (classe *Team*), plusieurs patients (classe *Patient*) et plusieurs prises en charge (classe *Management*). Toutefois, une prise en charge ne concerne qu'un seul patient et n'est réalisée que par une seule équipe et doit être réalisée dans le cadre d'une intervention bien déterminée. Lors d'une prise en charge d'un patient par une équipe, plusieurs actes de soin (classe *ManagementAct*) peuvent êtres réalisés. Ces derniers sont de différentes natures : *MedicalAdvice*, *Diagnosis*, *Care* etc.

Chapitre 2

Plateforme de transformation

Ce chapitre met l'accent sur les principes que nous avons mis en oeuvre en vue d'automatiser le processus de transformation du modèle fonctionnel Res@mu en des notations formelles B. Notons que les principes présentés ici ont été publiés à la conférence AFADL'10 (Approches Formelles dans l'Assistance au Développement de Logiciels).

La motivation majeure de ce travail provient, d'un côté, du fait que dans la littérature peu d'efforts ont été consacrés à l'outillage des approches de traduction d'UML en B ainsi qu'à leurs évolutions ; et d'un autre côté, de la nécessité de disposer d'un outil utilisable et applicable sur de vrais systèmes. Ainsi, dans le cadre de nos travaux autour du projet Selkis, nous proposons une plateforme basée sur les techniques de l'IDM et permettant de supporter les approches d'intégration de formalismes. Notre plateforme se veut générique en permettant d'intégrer plusieurs approches transformationnelles d'UML en des notations formelles. Elle se veut également évolutive en permettant, d'une part, d'étendre et de combiner les transformations mises en œuvre et d'autre part, d'exprimer des règles de transformation dans divers langages (*e.g.* Java, xTend, etc). Dans ce chapitre, nous proposons un aperçu des principes fondateurs de notre plateforme. Le chapitre suivant montre les résultats obtenus grâce à cet outillage.

2.1 Introduction

L'intégration d'approches formelles et semi-formelles permet une utilisation plus abordable des notations formelles et plus précise des notations semi-formelles. Bien que ce sujet soit largement traité dans la littérature, son application dans la pratique reste limitée. Comme l'explique [KBC05], l'un des obstacles est le flou sémantique entourant ces approches et qui est dû au fait que les transformations entre notations sont souvent définies implicitement. Le manque d'outils pour supporter ces techniques met, également, un frein considérable à leur adoption.

La solution émergente à cette problématique, comme proposé par [KBC05, GMK+06, ZJBZ08, NOW09], prêche l'utilisation de l'Ingénierie Dirigée par les Modèles (ou IDM) pour définir et mettre en œuvre les transitions entre langages. En effet, ce paradigme requiert une définition explicite des transformations sur la base de méta-modèles définissant les modèles sources et cibles et d'un langage de transformation de modèles. L'avantage majeur en est de disposer d'un cadre conceptuel clair, de par l'utilisation des méta-modèles, pour fonder les transformations d'un langage en un autre. De ce fait, les règles de transformation qui en découlent sont souvent qualifiées de précises. Aussi, offrir une plateforme IDM pour supporter l'activité d'intégration d'UML et de méthodes formelles, est-il un moyen pragmatique pour contribuer à populariser le couplage de méthodes. Toutefois, l'implication de la communauté (aussi bien scientifique qu'industrielle) reste le moteur principal pour diffuser ces approches et relever les défis restants avant le passage à l'échelle. Pour ce faire, disposer d'une plateforme de transformation commune

qui permet l'implémentation et l'expérimentation d'approches diverses, devient de plus en plus incontournable.

Dans le cadre de nos travaux de recherche nous abordons cette question en mettant en oeuvre une plateforme IDM qui se doit d'être évolutive pour supporter une multitude d'approches de traduction d'UML en des notations formelles. Cette plateforme se veut ouverte en permettant l'intégration de plusieurs langages de modélisation et de plusieurs transformations entre ces langages. Notre moteur de transformation permet de traiter une spécification abstraite d'un processus de transformation en faisant le lien avec les règles concrètes qui en découlent. L'originalité de ce travail se traduit par la possibilité de faire cohabiter plusieurs approches de transformation d'UML au sein d'un même environnement avec des règles de transformation pouvant être exprimées dans des langages variés. Cela permet une grande souplesse dans l'expression des transformations d'une part, et l'évolution et la personnalisation des approches existantes d'autre part.

2.2 Utilisation de l'IDM pour l'intégration de méthodes

Une architecture IDM est souvent une architecture à trois composantes qui sont outillées par trois modules intégrables à la plateforme EMF[1] : deux modules pour l'édition des modèles sources et cibles et un module d'expression et d'exécution de règles de transformation. L'idée principale de ce type d'architecture est de définir dans un langage de transformation la transition entre un modèle d'entrée et un modèle de sortie. Dans le cadre de nos travaux une telle architecture nous permettra de disposer d'un socle unique proposant la modélisation UML, la modélisation formelle et la transformation de modèles. Notre plateforme IDM est donc constituée de :

- Un module de modélisation UML avec le méta-modèle correspondant et un éditeur de diagrammes pour pouvoir manipuler graphiquement les modèles. Plusieurs éditeurs UML existent en tant que plugins Eclipse (TopCasedUML, Papyrus, etc). Dans notre plateforme nous avons intégré la spécification de UML2.1 disponible dans la distribution "Modeling" d'Eclipse [EMP10]. Cette dernière fournit entre autres un éditeur de diagrammes UML utile pour visualiser et éditer les modèles UML.
- Un module de modélisation formelle avec le méta-modèle correspondant et un éditeur de syntaxe concrète. Pour le cas des transformations d'UML en B nous avons considéré le méta-modèle de B proposé par [Ida06][2]. Pour passer de cette syntaxe abstraite de B vers la syntaxe concrète, nous avons intégré une amélioration de l'outil de génération de code proposé dans [IBP09].
- Un module de transformation de modèles dans lequel nous pouvons définir et exécuter les transformations entre UML et les notations formelles. Ce module constitue le cœur de notre travail et sera présenté en détail dans la suite du chapitre. Notons qu'il existe une diversité d'outils/langages de transformation de modèles compatibles avec la plateforme EMF d'Eclipse. Toutefois ils n'offrent pas les mêmes fonctionnalités et sont basés sur des paradigmes différents. Le choix du langage de transformation dépend des besoins de la transformation et de l'expérience de l'utilisateur.

[1] Eclipse Modeling Framework

[2] Ce méta-modèle (ecore) est disponible sous http ://membres-liglab.imag.fr/idani/BMeta/BMethod.ecore

2.3 Le module de transformation

2.3.1 Aperçu et contributions

Classiquement, dans les environnements IDM, le module de transformation est composé d'un moteur de transformation qui interagit avec les deux autres modules en vue de réaliser la transformation d'un modèle source vers un modèle cible. Cependant, cela suppose qu'une transformation est encodée de façon statique et il devient par conséquent difficile de changer les règles utilisées dans une transformation. Ce besoin se ressent fortement quand l'analyste est face à plusieurs solutions alternatives lors de la transformation d'un même élément source. Par exemple, en parcourant les divers travaux qui ont cherché à définir des règles de transformation d'un modèle objet en un modèle relationnel, nous avons constaté que certains désirent obtenir autant de tables relationnelles que de classes dans le modèle objet, tandis que d'autres ne produisent des tables que pour les classes concrètes du modèle. Pour ce faire, dans les environnement IDM existants, il faut agir directement sur le code de la transformation et proposer autant de blocs de règles que d'alternatives. Ainsi, l'intérêt de notre module de transformation en comparaison avec les modules existants est qu'il permet de décomposer une transformation en règles élémentaires (dites abstraites) qui pourront être instanciées par diverses règles concrètes . L'utilisateur pourra par la suite sélectionner la séquence de règles souhaitée. Dans notre approche, une transformation de modèles devient configurable et le choix d'une solution parmi d'autres n'est autre qu'une configuration particulière de la transformation.

Rappelons que notre objectif principal est de pouvoir, d'une part, combiner aisément différentes règles issues d'approches diverses et d'autre part, d'étendre ces approches par de nouvelles règles. Cela nécessite la composition de règles pour pouvoir définir une transformation configurable selon un choix de règles déterminé par l'utilisateur. L'originalité de notre approche est, de ce fait, l'introduction d'une couche supplémentaire permettant de spécifier, au moyen d'un métamodèle, la notion de transformation configurable.

Outre le fait que notre plateforme permet à l'utilisateur de sélectionner des règles diverses provenant d'approches variées, les règles elles-mêmes peuvent être codées dans divers langages de transformation. Cela procure non seulement une souplesse dans le choix de la transformation mais également une souplesse dans l'implémentation des transformations. Notre moteur de transformation interprète une configuration de transformation et délègue l'exécution des règles concrètes à d'autres outils de transformation. Pour ce faire, la transformation est guidée par un processus de transformation abstrait. L'outil de transformation que nous proposons ne se présente donc pas sous la forme d'un langage de transformation assisté par un moteur de transformation. En effet, notre intention n'est pas de proposer un nouveau language, mais plutôt de développer un invocateur de moteurs de règles (*e.g.* ATL [JABK08] ou QVT [OMG08], ...) existants en vue d'exécuter les règles référencées dans notre module de transformation lors d'une configuration de transformation.

2.3.2 Le processus de transformation

La notion de processus abstrait permet de subdiviser une transformation de modèle en un ensemble de phases séquentielles dont le comportement est variable selon l'implémentation. Chacune de ces phases, que nous appelons règle abstraite, transforme un type d'élément donné. Le vrai comportement de ces règles, notamment les éléments produits, dépend des implémentations qui les réalisent. En effet, une même règle abstraite peut être implémentée de plusieurs manières différentes et dans des langages de transformation différents. Ces implémentations sont appelées règles concrètes. Ainsi, chaque règle abstraite est réalisée par une ou plusieurs règles concrètes. Par exemple, transformer un diagramme de classes, comme l'illustre la figure 2.1, revient à transformer successivement les paquetages, les classes, les associations, les attributs et les opérations

de ce diagramme.

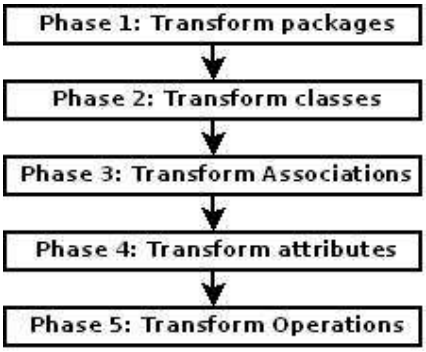

FIG. 2.1 – Un processus abstrait pour la transformation d'un diagramme de classes UML.

2.3.3 Spécification/exécution des transformations

La configurabilité des transformations et le méta-modèle de transformations configurables correspondant ont été discutés dans [ILL10] ; nous n'allons donc pas les reprendre en détail dans le présent chapitre. La figure 2.2 en présente un fragment utile pour la suite. Au niveau de ce méta-modèle de configuration, une transformation de modèle est définie par trois types d'entités : (a) les règles abstraites, (b) les types d'éléments sources et (c) les règles concrètes.

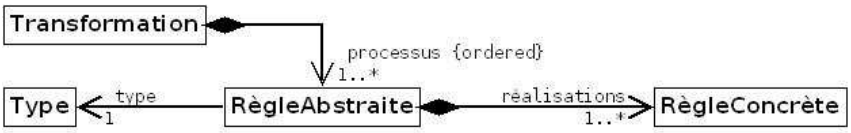

FIG. 2.2 – Fragment du méta-modèle de transformation configurable.

Une transformation est spécifiée par un ensemble ordonné de règles abstraites que nous appelons processus abstrait. Chacune de ces règles est associée au type d'éléments du modèle d'entrée qu'elle transforme. Elle est, également, associée à un ensemble (non vide) de règles concrètes qui la réalisent. Les types d'éléments sources sont les types d'éléments transformés par les règles abstraites et définis dans le méta-modèle d'entrée (e.g. Class, Association, etc.). La figure 2.3 présente globalement l'architecture de notre module de transformation. On y voit le moteur de transformation qui fait appel à trois autres composants (interfaces). En effet, le moteur de transformation interprète un modèle de transformation configurable (CTM) pour transformer un modèle d'entrée (InputModel) en un modèle de sortie (OutputModel). De plus, il stocke les liens entre le modèle d'entrée et le modèle de sortie dans un modèle de trace (TraceModel). L'ensemble des modèles manipulés par le moteur de transformation, que nous appelons contexte de la transformation, sont regroupés au sein du composant *TransformationContext*. Le moteur de transformation reconnaît les types des éléments en entrée grâce à une interface (appelée *InstanceProvider*) et peut de ce fait en extraire automatiquement les instances pour les transformer. L'implémentation par défaut de l'interface *InstanceProvider* ne reconnaît que les types prédéfinis dans le méta-modèle d'entrée. Ainsi, pour prendre en compte de nouveaux types en entrée il suffit d'étendre le méta-modèle d'entrée et de surcharger par conséquent l'*InstanceProvider*.

Contrairement aux règles abstraites, les règles concrètes sont des règles exécutables. Elles permettent de réaliser les règles abstraites auxquelles elles sont associées. Notre outil de transformation n'offre pas de langage pour l'implémentation de ces règles mais propose la réutilisation des langages et outils existants. En effet, les règles concrètes permettent de référencer, au moyen d'identifiants, des règles externes implémentées avec des outils tiers. Concrètement, le moteur

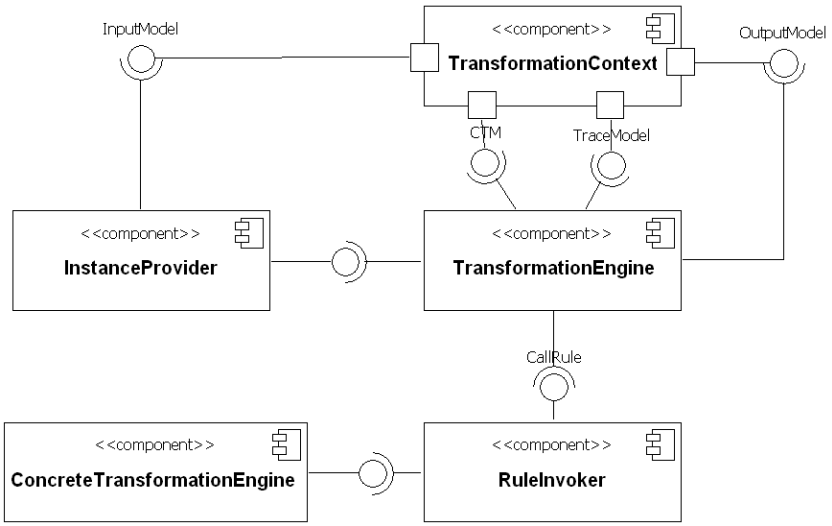

FIG. 2.3 – Architecture du module de transformation.

de transformation récupère la référence de la règle externe à appliquer et l'exécute à l'aide d'une interface particulière appelée invocateur de règles (*RuleInvoker*). L'implémentation actuelle de *RuleInvoker* permet d'invoquer des règles écrites en tant que méthodes Java ou en tant qu'extensions (opérations) en xTend[3] à travers leurs API respectives (ConcrèteTransformationEngine). Nous envisageons, en perspective, de l'étendre à d'autre langages comme QVT. Cette dissociation entre règles abstraites et règles concrètes (spécification/exécution) permet de faire varier dynamiquement le comportement des règles de la transformation et ainsi le comportement global de la transformation. Le moteur de transformation déterminera dynamiquement, à partir des paramètres de configuration, la règle concrète qui réalisera chacune des règles abstraites.

2.3.4 Configuration de la transformation

Pour un processus de transformation, la réalisation de chaque règle abstraite devient un point de décision qu'il faut résoudre en fonction de la configuration. Pour ce faire, nous proposons deux méthodes (manuelles) de configuration pour guider ces décisions : (*i*) la méthode de sélection par **règle** et (*ii*) la méthode de sélection par **approche**.

La première consiste à référencer explicitement la règle concrète à utiliser pour chaque règle abstraite. Cette méthode offre une grande flexibilité dans la configuration de la transformation. Néanmoins, l'application de cette méthode devient difficile lorsque le nombre de points de décision augmente considérablement. Pour remédier à ce problème, nous proposons une deuxième méthode plus intuitive pour l'utilisateur. Il s'agit de permettre à l'utilisateur de pré-définir des combinaisons de règles concrètes produisant des résultats différents selon les besoins. Ces combinaisons pré-définies que nous appelons **approches de transformation** correspondent généralement à des critères de classification du résultat obtenu. Par exemple, pour le cas de la traduction d'UML vers B [LM00a, Led01, SB06] l'un des critères utilisable est la modularité de la spécification B obtenue. Nous pouvons, par exemple, proposer trois approches de transformation d'un diagramme de classes basées sur ce critère :

(*i*) génération d'une seule machine B pour l'ensemble du diagramme de classes,
(*ii*) génération d'une machine par classe UML ou
(*iii*) génération d'une machine par paquetage.

Les deux méthodes de configuration sont, en fait, complémentaires. En effet, la méthode de sélection par règle permet de personnaliser les approches existantes tandis que la méthode de

[3] langage de transformation de modèles de l'outil oAW

sélection par approche permet de pérenniser ces approches et d'en proposer de nouvelles dont l'intention est d'être réutilisées ultérieurement sur d'autres modèles.

Nous distinguons en plus des deux méthodes de configuration deux niveaux pour leur application. Le premier niveau — que nous appelons niveau **méta-modèle** ou $\mathcal{M}^2$ par référence aux quatre niveaux de méta-modélisation de l'[OMG09, section 7.10] — correspond à l'application de la configuration uniformément sur l'ensemble du modèle source. C'est à dire que le choix de la règle concrète qui réalisera chacune des règles abstraites est récupéré exclusivement à partir des paramètres de configuration. Concrètement, tous les éléments du modèle d'entrée ayant le même type seront traduits par la même règle concrète désignée dans la configuration et produiront donc le même type de résultat. Ce niveau d'application est connu sous le nom de Model Type Mapping dans la spécification MDA de l'[OMG03, section 3.4].

Le deuxième niveau, que nous appelons niveau **modèle** ou $\mathcal{M}^1$, permet de personnaliser la transformation du niveau $\mathcal{M}^2$ en forçant certains éléments du modèle en entrée à être transformés de façon différente de la configuration. Nous proposons pour cela le marquage de ces éléments au moyen de stéréotypes indiquant la règle à leur appliquer.

2.4 Discussion

Une multitude de travaux de recherche se sont intéressés à l'intégration de UML et de notations formelles et ont abouti à diverses règles de transformation, souvent complémentaires. Cependant, et bien que ces recherches disposent de plusieurs bénéfices provenant de l'apport des méthodes formelles et semi-formelles, elles restent dédiées à des cadres applicatifs bien spécifiques. Par conséquent, elles ne sont pas applicables à plus grande échelle. Dans nos travaux actuels, nous proposons une plateforme IDM, d'intégration d'UML et de langages formels, qui se veut générique de part le fait qu'elle peut s'adapter à des approches transformationnelles diverses. L'avantage en est de pouvoir faire cohabiter, dans un même environnement des règles de transformation provenant d'approches complémentaires, ce qui permettra de les étendre et de les personnaliser. Ce chapitre a présenté notre plateforme en prenant le cadre particulier des approches de couplage de UML et B. Cependant, il est à noter que notre contribution s'inscrit dans un contexte plus global et peut s'appliquer au couplage d'UML et d'autres langages formels (comme Z).

Nous avons distingué les notions de règles abstraites et concrètes qui permettent d'avoir une souplesse dans la définition du processus de transformation d'UML. Des règles concrètes différentes peuvent implémenter la même règle abstraite. Outre ces contributions liées à la spécification de la transformation, notre plateforme permet de paramétrer et de configurer les transformations selon le niveau d'abstraction $\mathcal{M}^2$ ou $\mathcal{M}^1$. Le niveau $\mathcal{M}^2$ permet l'application uniforme de l'ensemble du processus de transformation sur tous les éléments UML en entrée, alors que le niveau $\mathcal{M}^1$ permet de forcer l'application de certaines règles pour des éléments particuliers du modèle. Cela permet de combiner des règles alternatives provenant d'approches diverses.

Chapitre 3

Spécfications B du modèle Res@mu

3.1 Introduction

Dans ce chapitre nous présentons les spécifications B générées par notre outil à partir du diagramme de classes de la figure 1.9 ainsi que les scénarios que nous avons mis en place en vie de valider le modèle fonctionnel. Rappelons que le processus de transformation que nous avons proposé dans le chapitre précédent prend en compte des transformations multiples du modèle UML. Ce processus est récapitulé par la figure 3.1 où une transformation est appelée “configuration”.

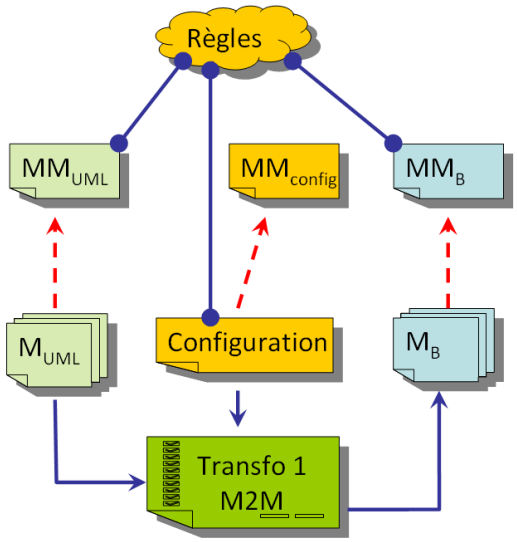

Fig. 3.1 – Processus de transformation

Notre outil permet de spécifier autant de configurations que de transformations possibles en instanciant un méta-modèle dédié à cela (MM_{config}). En revanche, en vue de pouvoir proposer une transformation en B du modèle de sécurité nous sommes contraints de choisir une transformation bien déterminée du modèle fonctionnel.

Les spécifications B issues du diagramme de classes de la figure 1.9 font presque 1500 lignes de code et produisent 568 obligations de preuve. Dans la suite de ce chapitre nous essayerons de nous focaliser sur les concepts les plus importants issus de ces spécifications.

3.2 Principes de la traduction

3.2.1 Traduction du diagramme de classes

Dans le chapitre 2 nous avons montré qu'un diagramme de classes peut donner lieu à une ou plusieurs machines B. L'idée d'avoir plusieurs machines B a été proposée par [LM00b] et est motivée par le fait de garder au niveau des spécifications B une structuration proche des classes du diagramme de classes. Cependant, cela implique diverses dépendances qui peuvent s'avérer complexes à gérer sur le plan conceptuel car elles reflètent des associations entre classes, des compositions, de l'héritage, etc.

Étant donné que l'exemple que nous traitons ici implique diverses classes (15 classes) et de nombreux liens entre classes (13 associations et 3 liens d'héritage), nous faisons le choix d'une configuration de transformation qui permet de générer une machine B unique pour tout le diagramme de classes. De ce fait, les liens entre classes seront traduits par des relations entre variables, plutôt que des liens entre machines B. Cela permet d'alléger l'activité de validation que nous allons décrire dans la section 3.3.

3.2.2 Traduction des classes

Chaque classe donne lieu à un ensemble abstrait représentant l'ensemble des instances possibles et une variable représentant l'ensemble des instances effectives. Nous obtenons ainsi dans la clause SETS et VARIABLES ce qui suit :

SETS
PREHOSPITALACTOR
; *PATIENT*
; *MANAGEMENTACT*
; *MANAGEMENT*
; *TEAM*
; *TEAMMEMBERCHANGEMENT*
; *INTERVENTION*
; *PREESTABLISHEDTEAM*
; *PERSON*
...

ABSTRACT_VARIABLES
PreHospitalActor
, *Patient*
, *ManagementAct*
, *Management*
, *Team*
, *TeamMemberChangement*
, *Intervention*
, *PreEstablishedTeam*
, *Person*
, *Diagnosis*
, *MedicalAdvice*
, *Care*
...

Des invariants d'inclusion sont générés dans la clause INVARIANT permettant de faire les liens entre instances possibles et instances effectives (par exemple $Patient \subseteq PATIENT$).

L'héritage

Remarquons qu'aucun ensemble abstrait n'est produit pour les sous-classes *MedicalAdvice*, *Diagnosis* et *Care*. Celles-ci sont traduites uniquement par des variables. En effet, l'ensemble abstrait qui englobe ces variables est issu de leur super-classe *ManagementAct*. De ce fait, l'invariant de typage correspondant est :

...
$\wedge$ $Diagnosis \subseteq ManagementAct$
$\wedge$ $MedicalAdvice \subseteq ManagementAct$
$\wedge$ $Care \subseteq ManagementAct$

Cette inclusion est suffisante pour pouvoir répercuter les attributs de la super-classe ainsi que ses liens sur ces sous-classes. En effet, si $\mathcal{A}$ est un attribut de ManagementAct de type $\mathcal{T}$ alors $\mathcal{A}$ sera défini par : $\mathcal{A} \in ManagementAct \rightarrow \mathcal{T}$. Comme une sous-classe $\mathcal{C}$ de *ManagementAct* est définie par $\mathcal{C} \subseteq ManagementAct$ alors l'ensemble des valeurs de l'attribut $\mathcal{A}$ hérité dans $\mathcal{C}$ correspond à : $\mathcal{C} \lhd \mathcal{A}$.

En outre, l'héritage dans cet exemple impose à un *ManagementAct* d'être soit un *MedicalAdvice*, soit un *Diagnosis* soit un Care. Pour ce faire, l'outil génère l'invariant suivant :

$$\begin{array}{|l} Diagnosis \cap MedicalAdvice = \emptyset \\ \wedge\ Diagnosis \cap Care = \emptyset \\ \wedge\ MedicalAdvice \cap Care = \emptyset \end{array}$$

Classes abstraites

La classe *ManagementAct* ne peut pas être instanciée car elle correspond à une classe abstraite indiquant qu'on ne peut instancier que ses sous-classes. Pour représenter en B les classes abstraites, nous considérons l'invariant $ManagementAct = MANAGEMENTACT$ ainsi qu'une initialisation conforme à cet invariant. Cela permet de disposer d'opérations B de construction d'éléments de types *MedicalAdvice*, *Diagnosis* et *Care* à partir de *ManagementAct* tout en interdisant la modification de la variable *ManagementAct*. Par conséquent, notre outil ne produit aucune opération B jouant le rôle de constructeur d'instances de *ManagementAct*.

3.2.3 Traduction des attributs de classes et de leurs types

Dans le diagramme de classes les attributs des classes sont modélisés sous la forme suivante :

```
nom_Attribut : type_Attribut multiplicité protection
```

La clause `type_Attribut` désigne le type de l'attribut et fait référence à trois sortes de types :

1. Les types de base (*e.g.* integer et booléen) : ces types sont traduits directement dans des types pré-définis de B tels que $\mathbb{Z}$ et BOOL.
2. Les chaînes de caractères : elles sont représentées par un ensemble abstrait nommé STR et déclaré dans la clause SETS.
3. Les types énumérés : ils sont traduits par des ensembles énumérés.

La clause `multiplicité` est facultative et indique si l'attribut est optionnel ou obligatoire. En effet, dans le diagramme de classes, la distinction du caractère optionnel de l'attribut est réalisée par une multiplicité [0..1]. Les autres attributs auxquels aucune multiplicité n'est associée sont obligatoires.

Contrairement aux attributs optionnels, les attributs obligatoires doivent être initialisés lors de l'instanciation de la classe. Ainsi, les paramètres des opérations de construction d'instances de classes que nous produisons dans la spécification B incluent les paramètres d'initialisation des attributs obligatoires. Par exemple, l'attribut obligatoire *name* de la classe *Person* est pris en compte dans l'opération de création d'instances de la manière suivante :

$$\begin{array}{|l} \textbf{Person\_NEW}(Instance, Person\_\_nameValue) = \\ \quad \textbf{PRE} \\ \qquad Instance \in PERSON \wedge Instance \notin Person \wedge \\ \qquad Person\_\_nameValue \in STR \wedge Person\_\_nameValue \notin \textbf{ran}(Person\_\_name) \\ \quad \textbf{THEN}\ Person := Person \cup \{Instance\} \\ \qquad\qquad \| \ Person\_\_name := Person\_\_name \cup \{(Instance \mapsto Person\_\_nameValue)\} \\ \quad \textbf{END}\ ; \end{array}$$

Chaque attribut est traduit par une relation dont le domaine et le co-domaine sont respectivement la variable B qui représente la classe dans laquelle l'attribut est encapsulé et la traduction B du type de l'attribut :

$$\texttt{nom\_Attribut} \in \texttt{nom\_Classe} \leftrightarrow \texttt{type\_Attribut}$$

Cette relation est spécialisée par une fonction totale ($\texttt{nom\_Classe} \rightarrow \texttt{type\_Attribut}$) si l'attribut est obligatoire et par une fonction partielle ($\texttt{nom\_Classe} \nrightarrow \texttt{type\_Attribut}$) si l'attribut est optionnel.

La clause `protection` est facultative et est souvent égale à {readonly}. Cette information sur le diagramme de classes permet de guider le processus de génération des opérations de base. En effet, on associe deux opérations de base aux attributs : les getters (pour lire la valeur de l'attribut) et les setters (pour modifier la valeur de l'attribut). Les setters ne sont pas générés pour les attributs marqués par {readonly}. Nous avons fait ce choix pour éviter de générer des opérations qui permettent d'effectuer des modifications incontrôlées de certains d'attributs. Cela n'interdit pas au concepteur d'ajouter ultérieurement d'autres opérations de modification spécifiques à l'attribut.

Pour illustrer la traduction des attributs, nous prenons l'exemple de la classe PreHospitalActor dont les attributs sont traduits en B comme suit :

$$\begin{array}{l} PreHospitalActor\_\_endingDateTime \in PreHospitalActor \nrightarrow \mathbb{Z} \\ \wedge\ PreHospitalActor\_\_startingDateTime \in PreHospitalActor \rightarrow \mathbb{Z} \\ \wedge\ PreHospitalActor\_\_state \in PreHospitalActor \nrightarrow ResourceState \\ \wedge\ PreHospitalActor\_\_operator \in PreHospitalActor \rightarrow \mathbf{BOOL} \\ \wedge\ PreHospitalActor\_\_medicalRole \in PreHospitalActor \rightarrow MEDICALROLE \end{array}$$

3.2.4 Traduction des associations

Les principes de la traduction des associations ont été présentés dans le livrable 3.2. Nous rappelons simplement que dans le cas général, une association assos liant deux classes A et B est traduite par une relation $\mathcal{R}$ entre les ensembles des instances effectives issus de ces classes ($\mathcal{E}_A$ et $\mathcal{E}_B$) :

$$assos \in \mathcal{E}_A \ \ \mathcal{R} \ \ \mathcal{E}_B$$

Les spécialisations de la relation $\mathcal{R}$ dépendent des multiplicités des deux côtés de l'association assos. Par exemple, des multiplicités * et 1 respectivement du côté de A et de B, donnent lieu à une fonction totale $\rightarrow$ de $\mathcal{E}_A$ vers $\mathcal{E}_B$.

Ce principe a été implémenté dans notre plateforme de transformation et a été appliqué au modèle fonctionnel Res@mu. Cependant, certaines informations supplémentaires ont été ajoutées sur le diagramme de classes en vue de guider ce processus de transformation lors de la génération des opérations de base. Les opérations de base destinées à la manipulation d'une association R entre deux classes A et B sont de la forme suivante :

- Constructeurs de liens : A_AddB_In_R, B_AddA_In_R
- Destructeurs de liens : A_DeleteB_From_R, B_DeleteA_From_R
- Getters de liens : A_Getb, B_Geta

Les constructeurs et destructeurs de liens permettent d'ajouter ou de supprimer une instance de R entre une instance de A et une instance de B. Ces opérations doivent, dans certains cas, être manipulées avec précautions en intégrant certains contrôles particuliers. Par exemple, une prise en charge (instance de la classe *Management*) est définie pour un et un seul *Patient*.

C'est pourquoi une prise en charge, au moment de sa création, est associée à un patient unique. Pour ce cas précis, aucune opération de modification de ce lien ne doit être proposée par le système car cela permettra de modifier le patient d'une prise en charge, ou de disposer de prises en charge sans patient, etc. Pour remédier à cela, nous proposons de marquer les extrémités des associations pour lesquelles ces opérations de modification sont critiques, par le mot clé `readonly`. Nous obtenons ainsi dans la spécification B uniquement les opérations de base de lecture (A_Getb, B_Geta).

3.2.5 Les contraintes fonctionnelles

Les spécifications B générées par notre outil incluent les structures de données issues des classes (ensembles et variables) ainsi que les invariants de typage. Les contraintes structurelles doivent être intégrées manuellement. Elle impliquent un travail de validation des opérations de base en vue de montrer que celle-ci n'aboutissent pas à des violations de contraintes. En UML les contraintes d'un diagramme de classes, sont généralement exprimées en OCL. Dans le présent travail nous faisons le choix d'identifier ces contraintes, de manière informelle, et de les formaliser directement en B. Ci-dessous, nous donnons quelques exemples de ces contraintes.

Les interventions : la terminaison d'une intervention nécessite une date de fin qui doit être postérieure à la date de début.

$$
\begin{array}{l}
\forall\,(ii).(ii \in INTERVENTION \wedge\ ii \in Intervention \Rightarrow \\
\quad (\\
\quad\ ((Intervention\_\_endingDateTime[\{ii\}] \neq \emptyset\ ) \\
\qquad \Rightarrow (\ Intervention\_\_endingDateTime(ii) > Intervention\_\_startingDateTime(ii))) \\
\quad) \\
) \\
\\
\wedge\ (\ \mathbf{dom}(Intervention\_\_closed \rhd \{\mathbf{TRUE}\}) = \mathbf{dom}(Intervention\_\_endingDateTime)\ )
\end{array}
$$

Clôture d'une prise en charge : la fin d'une prise en charge, telle que présentée dans le diagramme de séquences de la figure 1.5, nécessite une date de fermeture postérieure à sa date de début, et une indication sur l'état du patient (mise à jour de l'attribut *endingPatientState*).

$$
\begin{array}{l}
\forall\,(mm).(\ mm \in MANAGEMENT \wedge\ mm \in Management \Rightarrow \\
\quad (Management\_\_endingDateTime[\{mm\}] \neq \emptyset \\
\\
\qquad \Rightarrow (Management\_\_endingDateTime(mm) > Management\_\_startingDateTime(mm)) \\
\quad) \\
) \\
\\
\wedge\quad (\mathbf{dom}(Management\_\_endingDateTime) = \mathbf{dom}(Management\_\_endingPatientState))
\end{array}
$$

Cohérence du lien entre Team, Management est Intervention : L'équipe qui réalise une prise en charge est l'une des équipes impliquées dans l'intervention à laquelle la prise en charge est associée. Dans l'invariant ci-dessous, la relation $A\_Team\_Management$, issue de l'association entre les classes *Management* et *Team*, permet de retrouver l'équipe chargée d'une prise en charge. La relation $A\_Intervention\_Team$ est issue de l'association entre les classes *Intervention* et *Team*, et permet de retrouver les équipes impliquées dans une intervention. La relation $A\_Management\_Intervention$ représente l'association entre *Management* et *Intervention* et permet de retrouver l'intervention à laquelle une prise en charge est associée.

$$
\begin{array}{l}
\forall\,(mm).(\ mm \in MANAGEMENT \wedge\ mm \in Management \Rightarrow \\
\qquad (A\_Team\_Management(mm) \in A\_Intervention\_Team^{-1}\ [\{A\_Management\_Intervention(mm)\}]\ ))
\end{array}
$$

Cohérence du lien entre Patient, Management est Intervention : Le patient d'une prise en charge est l'un des patients de l'intervention à laquelle la prise en charge est associée. Dans l'invariant ci-dessous, la relation $A\_Patient\_Management$, issue de l'association entre les classes *Management* et *Patient*, permet de retrouver le patient d'une prise en charge. La relation $A\_Intervention\_Patient$ est issue de l'association entre les classes *Intervention* et *Patient*, et permet de retrouver les patients impliquées dans une intervention.

$$\forall\,(mm).(\ mm \in MANAGEMENT \wedge mm \in Management \Rightarrow \\ (\ A\_Patient\_Management(mm) \in A\_Intervention\_Patient[\{A\_Management\_Intervention(mm)\ \}]))$$

Cohérence des attributs temporels : Plusieurs attributs du diagramme de classes font référence aux dates de création, de validation, de fermeture, etc. Par conséquent, nous avons ajouté plusieurs invariants en vue de garantir leur cohérence.

$$\forall\,(mm).(mm \in MANAGEMENT \wedge mm \in Management \Rightarrow \\ (Management\_\_startingDateTime(mm) > Team\_\_creationDateTime(A\_Team\_Management(mm)))\ ))$$

$$\wedge$$

$$\forall\,(mm).(mm \in MANAGEMENT \wedge mm \in Management \\ \wedge\ (A\_Team\_Management(mm) \in \mathbf{dom}(Team\_\_endOfMissionDateTime)) \Rightarrow \\ (Management\_\_startingDateTime(mm) < \\ Team\_\_endOfMissionDateTime(A\_Team\_Management(mm)))\ ) \\ )$$

$$\wedge$$

$$\forall\,(mm).(mm \in MANAGEMENT \wedge mm \in Management \wedge mm \in \mathbf{dom}(Management\_\_endingDateTime) \\ \wedge\ \ (A\_Team\_Management(mm) \in \mathbf{dom}(Team\_\_endOfMissionDateTime))\ \ \Rightarrow \\ (\ Management\_\_endingDateTime(mm) < \\ Team\_\_endOfMissionDateTime(A\_Team\_Management(mm))) \\ )$$

$$\wedge$$

$$\forall\,(ma).(ma \in MANAGEMENTACT \wedge ma \in ManagementAct \Rightarrow \\ (\ ManagementAct\_\_dateTime(ma)\ > \\ Management\_\_startingDateTime(A\_Management\_ManagementAct(ma))\ ) \\ )$$

$$\wedge$$

$$\forall\,(ma).(ma \in MANAGEMENTACT \wedge ma \in ManagementAct \\ \wedge\ A\_Management\_ManagementAct(ma) \in \mathbf{dom}(Management\_\_endingDateTime) \Rightarrow \\ (\ ManagementAct\_\_dateTime(ma)\ < \\ Management\_\_endingDateTime(A\_Management\_ManagementAct(ma))\ ) \\ )$$

$$\ldots$$

Validation des actes de soin : Un praticien qui valide un acte de soin doit renseigner une date de validation. Si l'acte de soin est invalidé alors il ne doit pas contenir de date de validation.

$$\mathbf{dom}(ManagementAct\_\_validationDateTime) = \mathbf{dom}(ManagementAct\_\_validated \rhd \{\mathbf{TRUE}\}\ )$$

$$\wedge\ (\mathbf{dom}(ManagementAct\_\_invalidationDateTime) \cap \mathbf{dom}(ManagementAct\_\_validationDateTime)) = \emptyset$$

Globalement, nous avons ajouté manuellement 26 contraintes invariantes dans les spécifications B générées par notre outil portant en grande partie sur les attributs temporels. D'autres contraintes, non mentionnées ci-dessus, font référence à la constitution des équipes et aux éventuels changement des équipes au moment des interventions.

3.2.6 Les opérations

Les spécifications B issues du diagramme de classes Res@mu incluent 100 opérations dont 92 sont générées automatiquement et correspondent à des opérations de base. Les opérations que nous avons ajoutées manuellement sont explicitées dans le diagramme de classes.

ManagmementAct_validate et ***ManagmementAct_invalidate*** : permettent respectivement la validation et l'invalidation d'un acte soin. Notons que tous les attributs de la classe *ManagementAct* sont marqués par {`readonly`}. En effet, nous considérons que les opérations de base de modification d'un acte de soin sont critiques et que celles-ci doivent être ajoutées manuellement aux spécifications B avec attention en respectant les diverses contraintes fonctionnelles.

ManagementAct__validate(*Instance*,*time*)=
  PRE
    $Instance \in ManagementAct \quad \wedge \quad time \in \mathbb{Z}$
    $\wedge$ *ManagementAct__validated*(*Instance*)=**FALSE**
    $\wedge$ $Instance \notin$ **dom**(*ManagementAct__invalidationDateTime*)
    $\wedge$ $time >$ *ManagementAct__dateTime*(*Instance*)
THEN
  ManagementAct__validated(*Instance*) :=**TRUE**
  || **ManagementAct__validationDateTime**(*Instance*) :=*time*
  END ;

ManagementAct__invalidate(*Instance*,*time*,*reason*)=
  PRE
    $Instance \in ManagementAct \wedge time \in \mathbb{Z} \wedge reason \in STR$
    $\wedge$ $Instance \notin$ **dom**(*ManagementAct__invalidationDateTime*)
    $\wedge$ *ManagementAct__validated*(*Instance*)=**FALSE**
    $\wedge$ $time >$ *ManagementAct__dateTime*(*Instance*)
  THEN
    ManagementAct__invalidationDateTime(*Instance*) := *time*
    || **ManagementAct__invalidationReason**(*Instance*) :=*reason*
  END ;

Management_endManagement : permet de terminer une prise en charge conformément au scénario 1.5 en mettant à jour les attributs *endingDateTime* et *endingPatientState*. Cette opération s'applique sur des prises en charge qui ne sont pas déjà clôturées et doit vérifier :

- que la date de fermeture est postérieure à la date de création de la prise en charge,
- que tous les actes de soins de la prise en charge ont été créés avant la date de fermeture de la prise en charge,
- que la fermeture de la prise en charge est effectuée avant de clôturer la mission de l'équipe qui réalise cette prise en charge.

Management__endManagment(*Instance*,*time*,*state*)=
  PRE
    $Instance \in Management \quad \wedge \quad time \in \mathbb{Z} \quad \wedge \quad state \in STR$
    $\wedge$ $Instance \notin$ **dom**(*Management__endingDateTime*)
    $\wedge$ $time >$ *Management__startingDateTime*(*Instance*)
    $\wedge$ (($Instance \in$ **ran**(*A_Management_ManagementAct*)) $\Rightarrow$
      ($time >$ **max**(*ManagementAct__dateTime*[*A_Management_ManagementAct*$^{-1}$ [{*Instance*}]])))
    $\wedge$ (($A\_Team\_Management(Instance) \in$ **dom**(*Team__endOfMissionDateTime*))
      $\Rightarrow$ ($time <$ *Team__endOfMissionDateTime*(*A_Team_Management*(*Instance*)))
)
  THEN
    Management__endingDateTime(*Instance*) := *time* ||
    Management__endingPatientState(*Instance*) :=*state*
  END ;

Intervention_Close : permet de clôturer une intervention en mettant à jour les attributs *closed* et *endingDateTime* de la classe *Intervention*. Cette opération s'applique sur des interventions qui n'ont pas été clôturées, et vérifie que :
– la date de fin est postérieure à la date de début,
– toutes les prises en charges de l'intervention ont été clôturées,
– toutes les équipes de l'intervention ont fini leurs missions,
– la clôture de l'intervention est effectuée après avoir renseigné les dates de fin de mission des équipes impliquées dans l'intervention.

Intervention__Close(*Instance*,*time*)=
PRE
Instance ∈ *Intervention* ∧ *time* ∈ ℤ
∧ *Intervention__closed*(*Instance*)=**FALSE**
∧ *time* > *Intervention__startingDateTime*(*Instance*)
∧ *A_Management_Intervention* $^{-1}$ [{*Instance*}] ⊆ **dom**(*Management__endingDateTime*)
∧ *A_Intervention_Team* $^{-1}$ [{*Instance*}] ⊆ **dom**(*Team__endOfMissionDateTime*)
∧ (*A_Intervention_Team* $^{-1}$ [{*Instance*}] ≠ ∅ ⇒
(*time* >**max**(*Team__endOfMissionDateTime*[*A_Intervention_Team* $^{-1}$ [{*Instance*}]])))
THEN
Intervention__closed(*Instance*) :=**TRUE** ||
Intervention__endingDateTime(*Instance*) :=*time*
END ;

Team_addMembers et ***Team_removeMembers*** : permettent respectivement d'ajouter et d'enlever un membre d'une équipe.

Team_end : permet d'indiquer la fin de mission d'une équipe.

Team_CreateTeamFromprestablishedOne : permet de créer une équipe à partir d'une équipe pré-établies.

Les opérations de manipulation des équipes sont relativement complexes vu les diverses contraintes invariantes qu'elles doivent respecter. Nous les présentons en annexes.

3.3 Elaboration de scénarios d'animation

Outre l'activité de preuve, nous avons essayé d'analyser les spécifications B du modèle fonctionnel en exploitant des scénarios d'animation. L'objectif est d'identifier des cas d'usage potentiellement intéressants lors de l'élaboration des règles de sécurité. Ces règles seront évoquées et analysées dans un prochain livrable.

Un scénario d'animation correspond à une suite d'appels d'opérations B. Ce faisant, en vue de pouvoir réaliser ces appels nous nous servons de l'animateur ProB [LB08] qui permet d'animer les opérations dont la pré-condition est évaluée à vrai. Cet outil indique d'une part l'état atteint suite à l'appel d'une opération, et identifie, d'autre part, les éventuelles violations d'invariants.

3.3.1 Choix pratiques

Valuations des ensembles

En vue de pouvoir réaliser l'animation de la spécification issue du modèle fonctionnel, nous devons disposer d'ensembles énumérés. Les structures B présentées dans la section 3.2.2 doivent donc être entièrement valuées. Pour ce faire, nous créons une machine B identique à celle présentée dans les sections précédentes que nous appellerons *initialisedSpec*. Celle-ci contient les structures B suivantes :

MACHINE
initialisedSpec

SETS
STR
; *TEAMNAME* ={*TEAMNAME1*,*TEAMNAME2*,*TEAMNAME3*,*PREESTABLISHED*}
; *PERSONNAME*
; *PREHOSPITALACTOR*={*TeamDoctor_*,*TeamNurse_*,*TeamRescuer_*,*Parm_*,*DrRegulator_*}
; *PATIENT*={*badshapePatient*,*drunkGuy*,*dyingPatient*}
; *MANAGEMENTACT*={*ACT1*,*ACT2*,*ACT3*,*ACT4*,*ACT5*,*ACT6*}
; *MANAGEMENT* = {*MANAGEMENT1*,*MANAGEMENT2*}
; *TEAM* ={*TEAM1*,*TEAM2*}
; *TEAMMEMBERCHANGEMENT*
; *INTERVENTION*={*INTERVENTION1*,*INTERVENTION2*}
; *PREESTABLISHEDTEAM*={*onePreEstablishedTeam*}
; *PERSON*={*TeamDoctor*,*TeamNurse*,*TeamRescuer*,*Parm*,*DrRegulator*,*speedyGnzales*}

; *ChangeType*={*IN_*,*OUT*}
; *ResourceState*={*AVAILABLE_IN_UNIT*,*AVAILABLE_OUTSIDE*,*INTERVENING*,*UNAVAILABLE*}
; *MEDICALROLE*={*DOCTOR*,*NURSE*,*PARM*,*NONE*,*RESCUER*}

Séquencement d'opérations

Par souci de clarté, et en vue de disposer d'une trace des opérations B que nous voulons animer, nous proposons de regrouper les appels d'opérations au sein d'une machine B à part. Cependant, en B le séquencement d'opérations ne peut être réalisé dans une machine abstraite. C'est pourquoi nous sommes amenés à réaliser ce séquencement dans un raffinement. Pour ce faire, nous créons une première machine abstraite, nommée *AbstractScenario*, et dans laquelle nous considérons qu'un scénario d'animation sera réalisé en deux étapes.

MACHINE
AbstractScenario
SETS
STATES = {*INIT*,*STEP1*,*STEP2*}
VARIABLES
state
INVARIANT
state ∈ *STATES*
INITIALISATION
state := *INIT*
OPERATIONS
step1 = **PRE** *state*=*INIT* **THEN** *state* :=*STEP1* **END**;
step2 = **PRE** *state*=*STEP1* **THEN** *state* :=*STEP2* **END**;
END

Les opérations abstraites step1 et step2 seront raffinées par la suite en vue d'inclure les séquences d'opérations souhaitées de *initialisedSpec*. Le choix de décomposer un scénario en deux étapes successives a pour objectif d'alléger la tache de ProB. En effet, lors de l'animation d'une suite d'opérations, ProB évalue toutes les pré-conditions de toutes les opérations. Cela peut s'avérer extrêmement coûteux, en terme d'espace mémoire, si on considère un espace d'état conséquent avec de nombreuses opérations à animer.

Gestion des attributs temporels

Dans le diagramme de classes, plusieurs attributs dont référence à des dates (création, clôture, validation, etc). La validation par animation que nous souhaitons réaliser ne vise pas la cohérence

de ces attributs mais plutôt des processus de violation de contraintes structurelles qui peuvent être dangereux du point de vue de la sécurité. Pour garantir la bonne initialisation de ces attributs, nous introduisons dans le raffinement de *AbstractScenario* une variable *clock* qu'on incrémente implicitement par un TIC et qu'on passera en paramètre aux opérations de mise à jours d'attributs temporels.

REFINEMENT
Scenario_1_simpleAdvice
REFINES
AbstractScenario
INCLUDES
initialisedSpec
CONSTANTS
TXT
PROPERTIES
$TXT \in STR$

VARIABLES
state,
clock
DEFINITIONS
$TIC == clock := 1 + clock$
INVARIANT
$state \in STATES$
$\wedge\ clock \in \mathbb{N}$
INITIALISATION
$state := FINAL$
$||\ clock := 0$

3.3.2 Scénario simple

Dans ce premier scénario nous considérons le cas le plus simple qui montre que la spécification B est opérationnelle et permet d'atteindre un état désiré. L'idée de ce scénario correspond au démarrage d'une intervention. Il s'agit de :
– créer une intervention,
– de créer une équipe (vide), et de l'associer à cette intervention,
– d'ajouter des membres à cette équipe (en l'occurrence un Parm),
– d'ajouter un patient à l'intervention (il s'agit ici de *drunkGuy*),
– de créer une nouvelle prise en charge dans l'intervention en question,
– et d'associer un acte de soin de type medicalAdvice à cette prise en charge.

step1 =
PRE
$state = INIT$
THEN
Intervention_NEW(*INTERVENTION1*,*clock*) ; **TIC** ;
Team_NEW(*TEAM1*,*INTERVENTION1*,*clock*,*TEAMNAME2*) ; **TIC** ;
Team__addMembers(*TEAM1*,{*Parm_*},*clock*) ; **TIC** ;
Intervention__AddA_Intervention_Patient(*INTERVENTION1*,*drunkGuy*) ; **TIC** ;
Management_NEW(*MANAGEMENT1*,*INTERVENTION1*,*drunkGuy*,*TEAM1*,*clock*) ; **TIC** ;
MedicalAdvice_NEW_Valid(*ACT1*,*MANAGEMENT1*,*Parm_*,*clock*,*TXT*) ; **TIC** ;
$state := STEP1$
END ;

L'animation de cette première étape du scénario avec ProB permet de créer une instance du diagramme de classes dans laquelle on dispose d'une intervention ayant une équipe, un patient et une prise en charge. Cette dernière comprend, pour ce patient, un acte de soin de type medicalAdvice. L'état de la machine B atteint suite à l'animation de cette première étape du scénario respecte l'invariant et correspond à ce qui suit :

```
ManagementAct={ACT1},
MedicalAdvice={ACT1},
Management={MANAGEMENT1},
Team={TEAM1},
Intervention={INTERVENTION1},
```

```
A_Intervention_Team={(TEAM1|->INTERVENTION1)},
A_Intervention_Patient={(INTERVENTION1|->drunkGuy)},
A_Management_Intervention={(MANAGEMENT1|->INTERVENTION1)},
A_Management_ManagementAct={(ACT1|->MANAGEMENT1)},
A_Patient_Management={(MANAGEMENT1|->drunkGuy)},
A_PreHospitalActor_ManagementAct={(ACT1|->Parm_)},
A_Team_PreHospitalActor={(Parm_|->TEAM1)},
A_Team_Management={(MANAGEMENT1|->TEAM1)},
A_team_AllPreHospitalActor={(TEAM1|->Parm_)},
ManagementAct__dateTime={(ACT1|->5)},
ManagementAct__validated={(ACT1|->TRUE)},
ManagementAct__validationDateTime={(ACT1|->5)},
MedicalAdvice__advice={(ACT1|->TXT)},
Management__startingDateTime={(MANAGEMENT1|->4)},
Team__creationDateTime={(TEAM1|->1)},
Intervention__startingDateTime={(INTERVENTION1|->0)},
Team__name={(TEAM1|->TEAMNAME2)},
TeamMemberChangement={TEAMMEMBERCHANGEMENT1},
A_Team_TeamMemberChangement={(TEAMMEMBERCHANGEMENT1|->TEAM1)},
A_PreHospitalActor_TeamMemberChangement={(TEAMMEMBERCHANGEMENT1|->Parm_)},
TeamMemberChangement__changement={(TEAMMEMBERCHANGEMENT1|->IN_)},
TeamMemberChangement__dateTime={(TEAMMEMBERCHANGEMENT1|->2)},
...
```

Les cinq dernières lignes sont relatives aux changements effectués sur une équipe. Dans le présent scénario, il s'agit d'un changement de type IN, réalisé à la date t (égale à 2) où Parm_ a rejoint TEAM1. Remarquons aussi que ACT1 apparaît dans les ensembles *ManagementAct* et *MedicalAdvice* et est validé lors de sa création (variable *ManagementAct__validated*).

La deuxième étape de ce scénario correspond à la fin de l'intervention. Cela se traduit par la clôture de la prise en charge, la fin de la mission de l'équipe et à la fermeture de l'intervention.

step2 =
PRE
  state = *STEP1*
THEN
  Management__endManagment(*MANAGEMENT1*,*clock*,*TXT*) ; **TIC** ;
  Team__removeMembers(*TEAM1*,{*Parm_*},*clock*) ; **TIC** ;
  Team__end(*TEAM1*,*clock*) ; **TIC** ;
  Intervention__Close(*INTERVENTION1*,*clock*) ; **TIC** ;
  state := *STEP2*
END

Cette deuxième étape aboutit aussi à un état satisfaisant les contraintes invariantes du modèle. Nous présentons ci-dessous les éléments les plus pertinents de cet état. Ces éléments correspondent aux différentes mises à jours des attributs temporels ainsi qu'à l'indication du changement effectué sur l'équipe TEAM1. En effet, il s'agit d'un changement de type OUT concernant l'acteur pré-hospitalier Parm_.

```
Management__endingDateTime={(MANAGEMENT1|->6)},
Team__endOfMissionDateTime={(TEAM1|->8)},
Intervention__endingDateTime={(INTERVENTION1|->9)},
A_PreHospitalActor_TeamMemberChangement =
            {(TEAMMEMBERCHANGEMENT1|->Parm_),(TEAMMEMBERCHANGEMENT2|->Parm_)},
Intervention__closed={(INTERVENTION1|->TRUE)},
TeamMemberChangement__changement=
            {(TEAMMEMBERCHANGEMENT1|->IN_),(TEAMMEMBERCHANGEMENT2|->OUT)},
TeamMemberChangement__dateTime=
            {(TEAMMEMBERCHANGEMENT1|->2),(TEAMMEMBERCHANGEMENT2|->7)},
```

3.3.3 Scénario plus abouti

Dans ce deuxième scénario nous considérons deux interventions, deux patients et deux équipes. L'intérêt de ce scénario est qu'il explore le fait qu'une même personne peut changer d'équipe et participer à des interventions différentes sans qu'elles ne soient clôturées. Bien que cela soit réalisable d'un point de vue fonctionnel, sur le plan de la sécurité, cela peut introduire des failles de sécurité. En effet, l'accès aux données d'un patient, ou la réalisation de prescriptions est conditionné par l'appartenance de la personne à l'équipe réalisant une prise en charge. C'est pourquoi ce changement d'équipe, doit être géré convenablement au niveau du filtre de sécurité.

step1 =
PRE
state = *INIT*
THEN
Intervention_NEW(*INTERVENTION1*,*clock*) ; **TIC** ;
Intervention__AddA_Intervention_Patient(*INTERVENTION1*,*dyingPatient*) ; **TIC** ;
Team_NEW(*TEAM1*,*INTERVENTION1*,*clock*,*TEAMNAME1*) ; **TIC** ;
Team__addMembers(*TEAM1*,{*DrRegulator_*},*clock*) ; **TIC** ;
Management_NEW(*MANAGEMENT1*,*INTERVENTION1*,*dyingPatient*,*TEAM1*,*clock*) ; **TIC** ;
Diagnosis_NEW(*ACT1*,*MANAGEMENT1*,*DrRegulator_*,*TXT*,*clock*) ; **TIC** ;
Intervention_NEW(*INTERVENTION2*,*clock*) ; **TIC** ;
Intervention__AddA_Intervention_Patient(*INTERVENTION2*,*badshapePatient*) ; **TIC** ;
Team_NEW(*TEAM2*,*INTERVENTION2*,*clock*,*TEAMNAME3*) ; **TIC** ;
state := *STEP1*
END

Dans cette première étape du scénario, l'équipe TEAM1 est chargée de l'intervention INTERVENTION1 concernant le patient dyingPatient. Dans le cadre de cette intervention l'acteur pré-hospitalier DrRegulator_, appartenant à TEAM1, effectue un diagnostic (instance de la classe Diagnosis) sur le patient dyingPatient. Ensuite, un deuxième patient est considéré (badshapePatient) dans le cadre d'une autre intervention nommée INTERVENTION2 et à laquelle l'équipe TEAM2 est associée.

Dans la seconde étape du scénario, DrRegulator_ quitte TEAM1, rejoint TEAM2, effectue un acte de soin de type MedicalAdvice sur le patient badshapePatient, et termine la prise en charge (MANAGEMENT2) de ce patient. Ensuite ce même acteur pré-hospitalier rejoint TEAM1 et réalise un autre acte de soin de type MedicalAdvice concernant le patient dyingPatient, et ce dans le cadre de l'intervention INTERVENTION1. Ensuite, l'acteur pré-hospitalier termine la prise en charge de dyingPatient et quitte TEAM1. Les missions de TEAM1 et TEAM2 sont ensuite terminées et les interventions INTERVENTION1 et INTERVENTION2 sont clôturées.

step2 =
PRE
state = *STEP1*
THEN
Team__removeMembers(*TEAM1*,{*DrRegulator_*},*clock*) ; **TIC** ;
Team__addMembers(*TEAM2*,{*DrRegulator_*},*clock*) ; **TIC** ;
Management_NEW(*MANAGEMENT2*,*INTERVENTION2*,*badshapePatient*,*TEAM2*,*clock*) ;
TIC ;
MedicalAdvice_NEW(*ACT2*,*MANAGEMENT2*,*DrRegulator_*,*clock*,*TXT*) ; **TIC** ;
Management__endManagment(*MANAGEMENT2*,*clock*,*TXT*) ; **TIC** ;
Team__removeMembers(*TEAM2*,{*DrRegulator_*},*clock*) ; **TIC** ;
Team__addMembers(*TEAM1*,{*DrRegulator_*},*clock*) ; **TIC** ;
MedicalAdvice_NEW(*ACT3*,*MANAGEMENT1*,*DrRegulator_*,*clock*,*TXT*) ; **TIC** ;
Management__endManagment(*MANAGEMENT1*,*clock*,*TXT*) ; **TIC** ;
Team__removeMembers(*TEAM1*,{*DrRegulator_*},*clock*) ; **TIC** ;
Team__end(*TEAM1*,*clock*) ; **TIC** ;
Team__end(*TEAM2*,*clock*) ; **TIC** ;
Intervention__Close(*INTERVENTION1*,*clock*) ; **TIC** ;
Intervention__Close(*INTERVENTION2*,*clock*) ; **TIC** ;
ManagementAct__validate(*ACT1*,*clock*) ; **TIC** ;
state := *STEP2*
END

Remarquons que ce scénario se termine par la validation de l'acte de soin ACT1 après la clôture de l'intervention. Il s'agit d'une validation à postériori ne pouvant être réalisée que par l'acteur pré-hospitalier ayant réalisé cet acte de soin. Au niveau du modèle fonctionnel, les rôles des utilisateurs ne sont pas exploitées et le système ne stocke aucune information sur la personne ayant réalisé l'acte de soin. Ces informations seront gérées au niveau du contrôle d'accès statique.

3.4 Conclusion

Dans ce chapitre nous avons évoqué les principes de la traduction du modèle fonctionnel Res@mu en B tout en mettant l'accent sur la distinction entre les parties automatisées et celles qui sont introduites manuellement, notamment les contraintes fonctionnelles et certaines opérations. Nous sommes partis d'un diagramme de classes contenant 15 classes, 13 associations et 3 liens d'héritage ; et nous avons obtenu une spécification B de 1500 lignes de code et produisent 568 obligations de preuve. À cette spécification B nous avons ajouté 26 contraintes invariantes et 8 opérations pour compléter les invariants de typage et les opérations de base générés automatiquement.

Nous avons effectué une première validation de ces spécifications via la preuve automatique de l'Atelier B, ainsi que l'élaboration de scénarios d'animation que nous avons exploités via l'outil ProB. Dans la suite du travail nous songeons à affiner cette validation en vue de préparer la mise en oeuvre du modèle de sécurité. Ce dernier sera présenté et étayé dans la prochaine mise à jour du présent livrable.

Chapitre 4

Conclusion générale

Ce livrable a présenté l'application des techniques de traduction d'UML vers B, que nous avons proposées, sur le modèle Res@mu. Nous avons outillé notre approche dans une plateforme IDM de multi-transformation capable de générer différentes spécifications B à partir du même modèle UML de départ. L'utilisation de notre outil a été guidée par un ensemble de choix pratiques concernant aussi bien le modèle Res@mu que les règles de transformation utilisées.

Choix concernant le modèle Res@mu : Le modèle Res@mu se présente sous la forme d'un ensemble de diagrammes de classes et de cas d'utilisation relatifs à une activité d'analyse. De ce fait, ces diagrammes ne sont pas suffisamment précis pour l'utilisation de l'outil. D'une part, parce qu'ils omettent certaines informations de modélisation tels que les rôles, certaines multiplicités, les types d'attributs..., et d'autre part, parce que les cas d'utilisation ne sont pas étayés par des scénarios. Pour remédier à cela nous avons sollicité des scénarios d'exécution relatifs à des entités potentiellement critiques. Cela nous a permis de cerner avec précision un sous-ensemble du diagramme de classes sur lequel nous avons pu appliquer avec succès notre outil.

Choix concernant les règles de transformation : Les spécifications B du modèle res@mu doivent être utilisables par le filtre de sécurité statique. Pour ce faire, nous avons été amenés à effectuer un choix de transformation dans lequel tous les éléments de modélisation du modèle fonctionnel sont intégrés au sein d'une même machine B. Les opérations de base ont été générées automatiquement et sont conformes aux règles décrites dans le livrable 3.2.

Les résultats que nous avons obtenus de notre application sont assez satisfaisants car ils nous ont permis, d'une part, d'effectuer une première validation du modèle fonctionnel via l'animation de scénarios d'exécution, et d'autre part, d'évaluer notre outil et de préparer ses futures évolutions. En effet, les paradigmes de l'ingénierie dirigée par les modèles mis en oeuvre dans l'outil nous ont été d'une grande utilité car ils nous ont permis de disposer d'un outil modulaire et évolutif. Cette évolution peut se traduire par l'évolution des méta-modèles et des règles de transformation mais également par l'étude de transformations multiples du filtre de sécurité.

A ce stade du travail, la traduction du modèle de sécurité n'est pas réalisée, car cela nécessite la finalisation de l'outil et son l'évolution pour prendre en compte certains cas particuliers lors de la traduction du modèle de sécurité en B. Cependant, la modélisation graphique des règles de sécurité est entamée et ne nécessite aucune adaptation supplémentaire de l'outil. L'application de notre outil sur le modèle de sécurité ainsi que sa validation formelle fera l'objet de la prochaine mise à jour de ce livrable.

Annexe A

Ensembles, variables et invariants de typage

MACHINE
ManagmentAct

SETS
STR
; *PREHOSPITALACTOR*
; *PATIENT*
; *MANAGEMENTACT*
; *MANAGEMENT*
; *TEAM*
; *TEAMMEMBERCHANGEMENT*
; *INTERVENTION*
; *PREESTABLISHEDTEAM*
; *PERSON*

;*ChangeType*={*IN_*,*OUT*}
;*ResourceState*={*AVAILABLE_IN_UNIT*,*AVAILABLE_OUTSIDE*,*INTERVENING*,*UNAVAILABLE*}
;*MEDICALROLE*={*DOCTOR*,*NURSE*,*PARM*,*NONE*,*RESCUER*}
ABSTRACT_VARIABLES
PreHospitalActor
, *Patient*
, *ManagementAct*
, *Management*
, *Team*
, *TeamMemberChangement*
, *Intervention*
, *PreEstablishedTeam*
, *Person*
, *Diagnosis*
, *MedicalAdvice*
, *Care*
, *A_Intervention_Team*
, *A_Intervention_Patient*

, $A\_Management\_Intervention$
, $A\_Management\_ManagementAct$
, $A\_Patient\_Management$
, $A\_PreHospitalActor\_ManagementAct$
, $A\_PreHospitalActor\_TeamMemberChangement$
, $A\_Team\_PreHospitalActor$
, $A\_Team\_TeamMemberChangement$
, $A\_Team\_Management$
, $A\_team\_AllPreHospitalActor$
, $A\_preEstablishedTeam\_preHospitalActor$
, $A\_preHospitalActor\_person$
, $PreHospitalActor\_\_endingDateTime$
, $PreHospitalActor\_\_startingDateTime$
, $PreHospitalActor\_\_state$
, $PreHospitalActor\_\_operator$
, $PreHospitalActor\_\_medicalRole$
, $Patient\_\_closed$
, $Diagnosis\_\_wording$
, $ManagementAct\_\_dateTime$
, $ManagementAct\_\_validated$
, $ManagementAct\_\_validationDateTime$
, $ManagementAct\_\_invalidationDateTime$
, $ManagementAct\_\_invalidationReason$
, $MedicalAdvice\_\_advice$
, $Management\_\_startingDateTime$
, $Management\_\_endingDateTime$
, $Management\_\_endingPatientState$
, $Team\_\_creationDateTime$
, $Team\_\_endOfMissionDateTime$
, $Intervention\_\_startingDateTime$
, $Intervention\_\_endingDateTime$
, $Intervention\_\_closed$
, $Patient\_\_data$
, $Patient\_\_name$
, $Team\_\_name$
, $TeamMemberChangement\_\_changement$
, $TeamMemberChangement\_\_dateTime$
, $Care\_\_data$
, $PreEstablishedTeam\_\_name$
, $Person\_\_name$

INVARIANT

$PreHospitalActor \subseteq PREHOSPITALACTOR$
$\wedge\ Patient \subseteq PATIENT$
$\wedge\ ManagementAct \subseteq MANAGEMENTACT$
$\wedge\ Management \subseteq MANAGEMENT$
$\wedge\ Team \subseteq TEAM$

$\land$ $TeamMemberChangement \subseteq TEAMMEMBERCHANGEMENT$
$\land$ $Intervention \subseteq INTERVENTION$
$\land$ $PreEstablishedTeam \subseteq PREESTABLISHEDTEAM$
$\land$ $Person \subseteq PERSON$
$\land$ $Diagnosis \subseteq ManagementAct$
$\land$ $MedicalAdvice \subseteq ManagementAct$
$\land$ $Care \subseteq ManagementAct$
$\land$ $A\_Intervention\_Team \in Team \rightarrow Intervention$
$\land$ $A\_Intervention\_Patient \in Intervention \leftrightarrow Patient$
$\land$ $A\_Management\_Intervention \in Management \rightarrow Intervention$
$\land$ $A\_Management\_ManagementAct \in ManagementAct \rightarrow Management$
$\land$ $A\_Patient\_Management \in Management \rightarrow Patient$
$\land$ $A\_PreHospitalActor\_ManagementAct \in ManagementAct \rightarrow PreHospitalActor$
$\land$ $A\_PreHospitalActor\_TeamMemberChangement \in TeamMemberChangement \rightarrow PreHospitalActor$
$\land$ $A\_Team\_PreHospitalActor \in PreHospitalActor \nrightarrow Team$
$\land$ $A\_Team\_TeamMemberChangement \in TeamMemberChangement \rightarrow Team$
$\land$ $A\_Team\_Management \in Management \rightarrow Team$
$\land$ $A\_team\_AllPreHospitalActor \in Team \leftrightarrow PreHospitalActor$
$\land$ $A\_preEstablishedTeam\_preHospitalActor \in PreHospitalActor \twoheadrightarrow PreEstablishedTeam$
$\land$ $A\_preHospitalActor\_person \in PreHospitalActor \rightarrow Person$
$\land$ $PreHospitalActor\_\_endingDateTime \in PreHospitalActor \nrightarrow \mathbb{Z}$
$\land$ $PreHospitalActor\_\_startingDateTime \in PreHospitalActor \rightarrow \mathbb{Z}$
$\land$ $PreHospitalActor\_\_state \in PreHospitalActor \nrightarrow ResourceState$
$\land$ $PreHospitalActor\_\_operator \in PreHospitalActor \rightarrow \mathbf{BOOL}$
$\land$ $PreHospitalActor\_\_medicalRole \in PreHospitalActor \rightarrow MEDICALROLE$
$\land$ $Patient\_\_closed \in Patient \rightarrow \mathbf{BOOL}$
$\land$ $Diagnosis\_\_wording \in Diagnosis \rightarrow STR$
$\land$ $ManagementAct\_\_dateTime \in ManagementAct \rightarrow \mathbb{Z}$
$\land$ $ManagementAct\_\_validated \in ManagementAct \rightarrow \mathbf{BOOL}$
$\land$ $ManagementAct\_\_validationDateTime \in ManagementAct \nrightarrow \mathbb{Z}$
$\land$ $ManagementAct\_\_invalidationDateTime \in ManagementAct \nrightarrow \mathbb{Z}$
$\land$ $ManagementAct\_\_invalidationReason \in ManagementAct \nrightarrow STR$
$\land$ $MedicalAdvice\_\_advice \in MedicalAdvice \rightarrow STR$
$\land$ $Management\_\_startingDateTime \in Management \rightarrow \mathbb{Z}$
$\land$ $Management\_\_endingDateTime \in Management \nrightarrow \mathbb{Z}$
$\land$ $Management\_\_endingPatientState \in Management \nrightarrow STR$
$\land$ $Team\_\_creationDateTime \in Team \rightarrow \mathbb{Z}$
$\land$ $Team\_\_endOfMissionDateTime \in Team \nrightarrow \mathbb{Z}$
$\land$ $Intervention\_\_startingDateTime \in Intervention \rightarrow \mathbb{Z}$
$\land$ $Intervention\_\_endingDateTime \in Intervention \nrightarrow \mathbb{Z}$
$\land$ $Intervention\_\_closed \in Intervention \rightarrow \mathbf{BOOL}$
$\land$ $Patient\_\_data \in Patient \rightarrowtail STR$
$\land$ $Patient\_\_name \in Patient \rightarrowtail STR$
$\land$ $Team\_\_name \in Team \rightarrowtail STR$
$\land$ $TeamMemberChangement\_\_changement \in TeamMemberChangement \rightarrow ChangeType$
$\land$ $TeamMemberChangement\_\_dateTime \in TeamMemberChangement \rightarrow \mathbb{Z}$
$\land$ $Care\_\_data \in Care \rightarrowtail STR$

$\land\ PreEstablishedTeam\_\_name \in PreEstablishedTeam \rightarrowtail STR$
$\land\ Person\_\_name \in Person \rightarrowtail STR$
$\land\ Diagnosis \cap MedicalAdvice = \emptyset$
$\land\ Diagnosis \cap Care = \emptyset$
$\land\ MedicalAdvice \cap Care = \emptyset$

Bibliographie

[EMP10] EMP. Eclipse modeling project, 2010.

[GMK+06] Jörn Guy Süß, Tim McComb, Soon-Kyeong Kim, Luke Wildman, and Geoffrey Watson. MDA-Based Re-engineering with Object-Z. In *International conference on Model Driven Engineering Languages and Systems (MoDELS)*, volume 4199 of *LNCS*, pages 291–305. Springer, 2006.

[IBP09] Akram Idani, Jean-Louis Boulanger, and Laurent Philippe. Linking paradigms in safety critical systems. *International Journal of Computers and their Applications. Special Issue on the Application of Computer Technology to Public Safety and Law Enforcement*, 16(2) :111–120, 2009.

[Ida06] Akram Idani. *B/UML : Mise en relation de spécifications B et de descriptions UML pour l'aide à la validation externe de développements formels en B.* PhD thesis, Université de Grenoble 1, November 2006.

[ILL10] Akram Idani, Mohamed-Amine Labiadh, and Yves Ledru. Infrastructure dirigée par les modèles pour une intégration adaptable et évolutive de UML et B. *RSTI - Ingénierie des Systèmes d'Information (ISI)*, 15(3), 2010.

[JABK08] Frédéric Jouault, Freddy Allilaire, Jean Bézivin, and Ivan Kurtev. ATL : A model transformation tool. *Sci. Comput. Program.*, 72(1-2) :31–39, 2008.

[KBC05] Soon-Kyeong Kim, Damian Burger, and David-A. Carrington. An MDA Approach Towards Integrating Formal and Informal Modeling Languages. In *International Symposium of Formal Methods (FM 2005)*, volume 3582 of *LNCS*, pages 448–464. Springer, 2005.

[LB08] Michael Leuschel and Michael J. Butler. ProB : an automated analysis toolset for the B method. *STTT*, 10(2) :185–203, 2008.

[Led01] Hung Ledang. Automatic translation from uml specifications to b. In *16th IEEE international conference on Automated software engineering*, page 436, 2001.

[LM00a] Régine Laleau and Amel Mammar. An Overview of a Method and Its Support Tool for Generating B Specifications from UML Notations. In *15th IEEE international conference on Automated software engineering*, pages 269–272, 2000.

[LM00b] Régine Laleau and Amel Mammar. An Overview of a Method and Its Support Tool for Generating B Specifications from UML Notations. In *15th IEEE International Conference on Automated Software Engineering*, pages 269–272, 2000. IEEE Computer Society Press.

[NOW09] Francisco Assis Nascimento, Marcio F. S. Oliveira, and Flávio Rech Wagner. Using MDE for the formal verification of embedded systems modeled by UML sequence diagrams. In *Proceedings of the 22nd Annual Symposium on Integrated Circuits and System Design : Chip on the Dunes*. ACM, 2009.

[OMG03] OMG. Mda guide version 1.0.1, 2003. www.omg.org/docs/omg/03-06-01.pdf.

[OMG08] OMG. Meta object facility (mof) 2.0 query/view/transformation specification version 1.0. http ://www.omg.org/spec/QVT/1.0/, April 2008.

[OMG09] OMG. *Unified Modeling Language : infrastructure (version 2.2)*. Object Management Group, February 2009. http ://www.omg.org/spec/UML/2.2/Infrastructure/PDF/.

[SB06] Colin Snook and Michael Butler. UML-B : Formal modeling and design aided by UML. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 15(1) :92–122, 2006.

[ZJBZ08] Tian Zhang, Frédéric Jouault, Jean Bézivin, and Jianhua Zhao. A MDE Based Approach for Bridging Formal Models. In *IFIP/IEEE International Symposium on Theoretical Aspects of Software Engineering (TASE)*, pages 113–116. IEEE CS Press, 2008.