jUCMNav: une nouvelle plateforme ouverte pour l'édition et l'analyse de modèles UCM

Jason Kealey, Etienne Tremblay, Jean-Philippe Daigle, Jordan McManus, Olivier Clift-Noël et Daniel Amyot

***Résumé*— La notation Use Case Map (UCM) permet de modéliser les fonctionnalités de divers types de systèmes réactifs et répartis sous forme de scénarios graphiques. L'outil *UCM Navigator*, développé au cours des huit dernières années, supporte la notation mais souffre de plusieurs problèmes de convivialité, de maintenance, de déploiement et d'évolutivité qui freinent l'utilisation à grande échelle de la notation et le développement de méthodes d'analyse robustes et efficaces. Cet article présente un nouvel outil, *jUCMNav*, basé sur la plateforme Eclipse. jUCMNav implémente un méta-modèle UCM à l'aide du « Eclipse Modeling Framework » et utilise le cadre de développement « Graphical Editing Framework » afin d'adresser les problèmes précédents. Cet article présente la conception de cette nouvelle plateforme et discute des possibilités qu'elle offre au niveau de la modélisation et de l'analyse d'exigences pour systèmes complexes. Un retour d'expérience sur son développement est aussi offert.**

I. Introduction

La notation *Use Case Map* (UCM) permet de modéliser les fonctionnalités de divers types de systèmes réactifs et répartis sous forme de scénarios graphiques [6]. Les exigences fonctionnelles peuvent être illustrées sous forme de scénarios démontrant visuellement une facette d'un système plus complexe (Fig. 1). Grâce à ses capacités d'abstraction et de réutilisation, la notation guide à la fois l'architecture du système et son comportement.

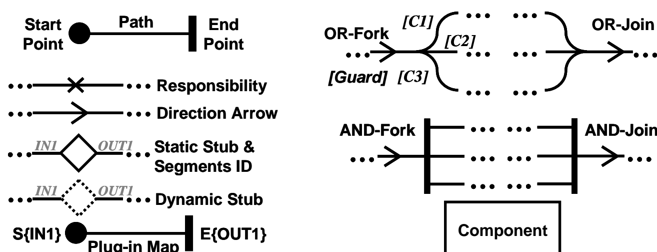

Fig. 1. Sommaire des éléments principaux de la notation UCM.

La notation UCM est aussi versatile; elle peut être utilisée pour la modélisation et la validation d'exigences [1][3] et de processus d'affaires [30], la génération de modèles de performance [23][32], la génération de scénarios pour la conception détaillée [2], la génération de tests [4] et la visualisation de programmes existants dans un contexte de rétro-ingénierie [10], le tout dans divers domaines d'application. La notation UCM partage plusieurs concepts retrouvés dans les diagrammes d'activités de UML 2.0 [21], mais permet elle permet aussi une meilleure description de certains aspects dynamiques de la structure et du comportement, elle intègre des aspects de performance et de test, et elle offre aussi un aspect visuel différent et plus évocateur au niveau de la combinaison scénarios-architecture. La notation UCM est aussi en cours de normalisation au sein de l'Union internationale des télécommunications (UIT) comme composante de la *User Requirements Notation* (URN) [1][27]. L'autre composante d'URN, appelée *Goal-oriented Requirement Language* (GRL), n'est pas discutée dans cet article.

L'outil *UCMNav* (pour *UCM Navigator*) [17], en constante évolution depuis 1997, permet de supporter les diverses applications de la notation via l'édition graphique de modèles UCM, leur analyse et leur transformation vers différents langages cibles (*Message Sequence Charts*, *Layered Queueing Networks*, diagrammes de séquence UML, formats de scénarios propriétaires, etc.). Il permet aussi depuis peu l'intégration de modèles UCM et de logiciels de gestion d'exigences, notamment avec *Telelogic DOORS* [15][25].

Les choix technologiques de l'époque ont porté l'outil à travers les années mais celui-ci se voit aujourd'hui difficile à déployer, loin d'être agréable à utiliser et, surtout, impossible à maintenir alors que la notation et ses applications évoluent. L'outil UCMNav actuel contient plus de 100 classes représentant 70 000 lignes de code C++ (peu documentées) où 10% du code est généré automatiquement par un éditeur d'interface utilisateur. L'architecture initiale a disparu sous le nombre de modifications et d'ajouts faits par une vingtaine de personnes (majoritairement des étudiants) sur une période de huit ans. Le couplage entre la gestion du modèle et l'interface est élevé à un point tel où les deux sont pratiquement indissociables. L'interface utilisateur est basée sur la librairie *XForms* [33] et l'outil requiert la présence d'un serveur *X Window* pour fonctionner [31]. Bien que ces choix aient permis à l'outil d'être disponible sur plusieurs systèmes d'exploitation (GNU/Linux, Solaris™, HP/UX™, MS Windows™ et Mac OS X™), le fait de nécessiter un serveur X est un obstacle majeur au déploie-

Les auteurs proviennent de l'École d'ingénierie et de technologie de l'information de l'Université d'Ottawa, 800 King Edward, Ottawa (Ontario), K1N 6N5, Canada (courriel: jkealey@shade.ca, damyot@site.uottawa.ca). Ce projet est supporté financièrement en partie par le programme de subventions stratégiques du CRSNG (Canada).

ment de l'outil sur la plateforme la plus commune, MS Windows™. L'interface utilisateur elle-même est non-standard, contre-intuitive, incomplète et contraignante. Étant donné l'absence d'une suite de tests accompagnant le code, l'outil est peu robuste et il est facile de briser des fonctionnalités existantes lors de modifications. La documentation sur l'architecture et les choix d'implémentation est aussi très limitée. La problématique affligeant UCMNav n'est pas hors du commun dans l'industrie, surtout en considérant le fait que cet outil n'était à la base qu'un prototype permettant d'explorer la notation et son utilisation.

Cet article présente une nouvelle plateforme ouverte pour l'édition et l'analyse de modèles UCM : *jUCMNav* (prononcé *juicy-emme-nav*). Ce nouvel outil, en cours de développement, cherche à combler les lacunes d'UCMNav en termes de convivialité, extensibilité, facilité de déploiement, facilité de maintenance, robustesse et documentation. L'article donne un aperçu de l'architecture de l'outil (lui-même basé sur la plateforme Eclipse), des choix technologiques et des fonctionnalités présentement offertes ou planifiées. Notre but est éventuellement d'offrir les meilleures fonctionnalités d'UCMNav sous un meilleur jour et de supporter la version standardisée de la notation UCM tout en offrant une base plus solide lors de l'évolution de la notation et de ses applications.

II. jUCMNav et Eclipse

Afin d'offrir une application portable et extensible, tout en profitant d'une quantité sans cesse croissante d'outils de développement logiciel de fine pointe, nous avons opté pour Java (1.5) comme langage d'implémentation. Java offre aussi un mécanisme de réflexion très utile pour gérer l'évolution d'outils de modélisation (la standardisation de la notation UCM n'étant pas encore complétée). Nous avons aussi opté pour Eclipse comme environnement de développement et d'exécution. Eclipse est une plateforme universelle pour le développement d'outils et, simultanément, un outil lui-même. La communauté `Eclipse.org` a vu le jour en 2001 avec la création du conseil d'administration formé de chefs de file de l'industrie (Borland, IBM, Red Hat et plusieurs autres). Étant un outil à code libre développé en Java et ayant comme but principal l'extensibilité, il permet aux développeurs d'étendre la plateforme pour y intégrer des éditeurs spécialisés. De par leur nature même, les extensions d'Eclipse (appelées *plug-ins*) se voient portables.

Le projet Eclipse comprend lui-même quatre sous-projets très pertinents pour jUCMNav (Fig. 2). Le plus connu et le plus visible est le *Java Development Tools* (JDT), un environnement de développement intégré permettant de développer rapidement des applications Java et des extensions pour Eclipse. Pour assurer la portabilité des interfaces utilisateur tout en conservant l'aspect de convivialité natif du système d'exploitation sous-jacent, le *Standard Widget Toolkit* (SWT) est alors utilisé par l'application.

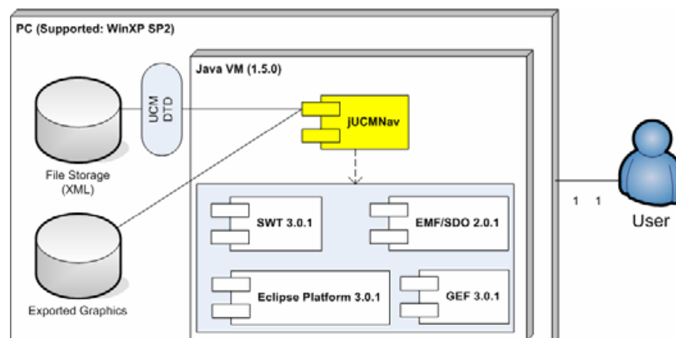

Fig. 2. Aperçu de l'architecture de haut niveau de jUCMNav.

Grâce au *Graphical Editing Framework* (GEF) [8], la plateforme est étendue pour offrir un modèle efficace pour le développement d'outils graphiques. Cette standardisation, couplée avec le respect des règles d'uniformité d'Eclipse, permet d'offrir une expérience d'utilisation riche et intuitive. Enfin, le *Eclipse Modeling Framework* (EMF) [9] propose un méta-méta-modèle permettant une gestion simplifiée de modèles, de même que la génération de code Java à partir de méta-modèles décrits par des diagrammes de classes UML [21]. Puisque les changements au méta-modèle utilisé pour décrire la syntaxe abstraite de la notation UCM sont reflétés automatiquement dans l'implémentation, l'évolution et la maintenance s'en voient facilités. EMF offre aussi la sérialisation et la lecture de modèles sous forme de fichiers XMI [22]. Enfin, GEF et EMF sont conçus pour coopérer étroitement dans le développement d'éditeurs graphiques en offrant des mécanismes de création, de gestion, de notification et de mise à jour selon le patron modèle-vue-contrôle (MVC) [18][20].

Un outil tel jUCMNav non seulement utilise les ressources de la plateforme Eclipse mais aussi étend cette dernière. L'outil devient alors une composante intégrée à l'environnement (en tant que *plug-in* Eclipse), cohabite avec d'autres plug-ins et permet même d'être étendu à son tour par d'autres plug-ins.

Notons aussi la grande popularité de la plateforme Eclipse, surtout auprès des concepteurs d'outils de génie logiciel. Plus de 395 plug-ins Eclipse (commerciaux et gratuits) sont listés sur `www.eclipseplugincentral.com`, l'un des catalogues de cette communauté. *Rational System Architect* est enfin un exemple de produit commercial basé sur EMF et GEF [13].

Pour toutes ces raisons, Eclipse nous est apparu comme un choix plus qu'intéressant pour le développement d'une nouvelle génération d'outils pour la modélisation UCM.

III. Méta-Modèle Use Case Map Réalisé sur EMF

A. Méta-Modèle UCM

La syntaxe abstraite et la sémantique de la notation UCM sont basées sur un méta-modèle défini par un ensemble de diagrammes de classes. Ceux-ci sont inclus dans la description de la *User Requirements Notation* [29]. Les paquetages pertinents et leurs dépendances sont représentés à la Fig. 3.

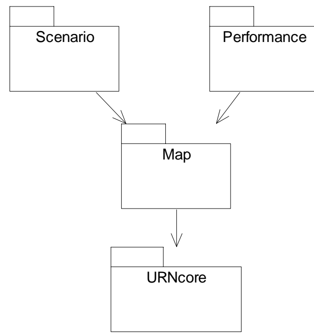

Fig. 3. Paquetages UML principaux pour la notation UCM.

URNcore définit quelques concepts de base tels que les responsabilités et les composantes. *Map* décrit la notation de base UCM alors que les paquetages *Scenario* et *Performance* spécifient les extensions permettant de décrire les définitions de scénarios et les annotations de performance pour diagrammes UCM. Le nouveau méta-modèle UCM a grandement été amélioré par rapport à ce qui se retrouvait dans UCMNav, notamment au niveau de l'intégration à GRL dans URN et au niveau des annotations de performance, mieux alignées avec celles définies dans [24][32].

Nous avons aussi étendu le méta-modèle UCM pour y inclure certaines informations graphiques telles que positions, tailles, couleurs, etc., ce qui complète l'information nécessaire pour décrire entièrement les diagrammes UCM.

Trois extraits du paquetage *Map* (le plus important paquetage de la Fig. 3) sont décrits afin de présenter sommairement la structure des diagrammes UCM. La première vue (Fig. 4) présente les différents types de nœuds (PathNode) retrouvés dans le graphe des chemins (PathGraph) composant un diagramme UCM (Map). On remarque entre autres des nœuds pour les débuts et fin de chemins, les chemins qui se séparent ou se rejoignent en alternative ou de façon concurrente, les boucles, les minuteries, les contenants pour sous-diagrammes (Stub), etc. Les différentes conditions et les politiques de sélection sont décrites à la Fig. 5.

La troisième vue (Fig. 6) présente la définition globale d'une spécification UCM (UCMSpec) composée d'un ensemble de diagrammes où les différents nœuds peuvent être inclus dans des composantes (ComponentRef), indiquant ainsi les relations qui existent entre comportement et architecture (c.-à-d. qui fait quoi). Les relations hiérarchiques qui existent entre les différents diagrammes UCM (PluginBinding) indiquent de quelle façon un sous-diagramme est connecté au stub de son diagramme UCM parent. Ces relations sont en général plus flexibles que celles permises par les diagrammes d'activité.

Sans entrer dans les détails, les autres diagrammes de classes qui ne sont pas présentés ici indiquent comment sont intégrées les définitions de scénarios (permettant d'extraire des scénarios individuels d'un modèle UCM complexe [2]), les annotations de performance [23][32], ainsi que les relations entre modèles UCM et modèles GRL au sein d'URN.

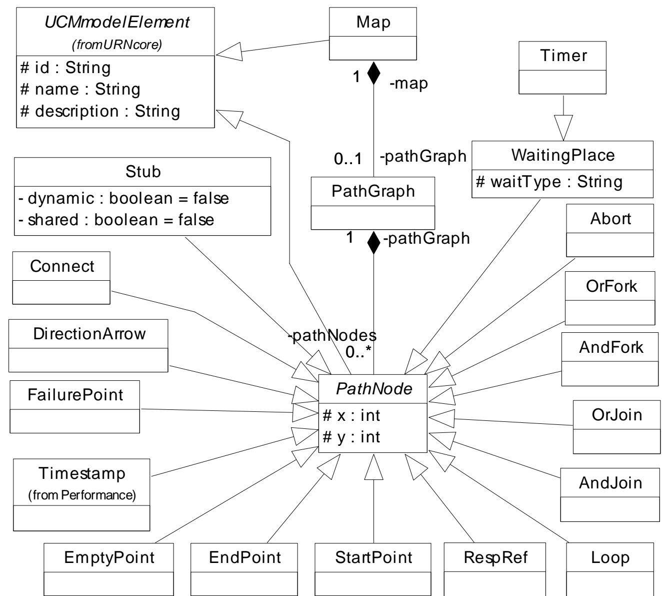

Fig. 4. Vue partielle du paquetage UCM : types de nœuds.

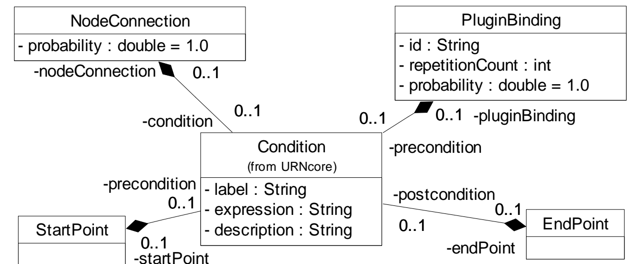

Fig. 5. Vue partielle du paquetage UCM : conditions.

Grâce à tous ces éléments, le concepteur d'un système distribué peut rapidement concrétiser les lignes directrices de sa pensée, puis en raffiner simultanément les aspects comportementaux et architecturaux.

B. Conversion vers EMF

Tel que mentionné à la section II, EMF permet la génération de code à partir de méta-modèles, qui peuvent être fournis de différentes façons, incluant les diagrammes de classes UML. Par exemple, nous avons utilisé Rational Rose [12] pour développer notre méta-modèle UCM, l'exporter en format XMI [22] et enfin l'importer sous Eclipse (en EMF).

À partir d'un méta-modèle EMF, trois différents ensembles de fichiers peuvent être générés. Premièrement, le méta-modèle en entier peut être converti en code Java, permettant aussi la sérialisation/persistance de modèles en fichiers XMI. Deuxièmement, EMF peut générer un ensemble de classes qui adapteront le méta-modèle pour la présentation. Finalement, la plateforme supporte la création d'éditeurs simples (en forme d'arbres) spécialisés pour notre méta-modèle.

Pour jUCMNav, plus de 50 000 lignes de code Java ont été générées par EMF pour représenter le méta-modèle, incluant une quantité importante de commentaires Javadoc (nous avons 38 000 lignes commentées pour le reste de l'application).

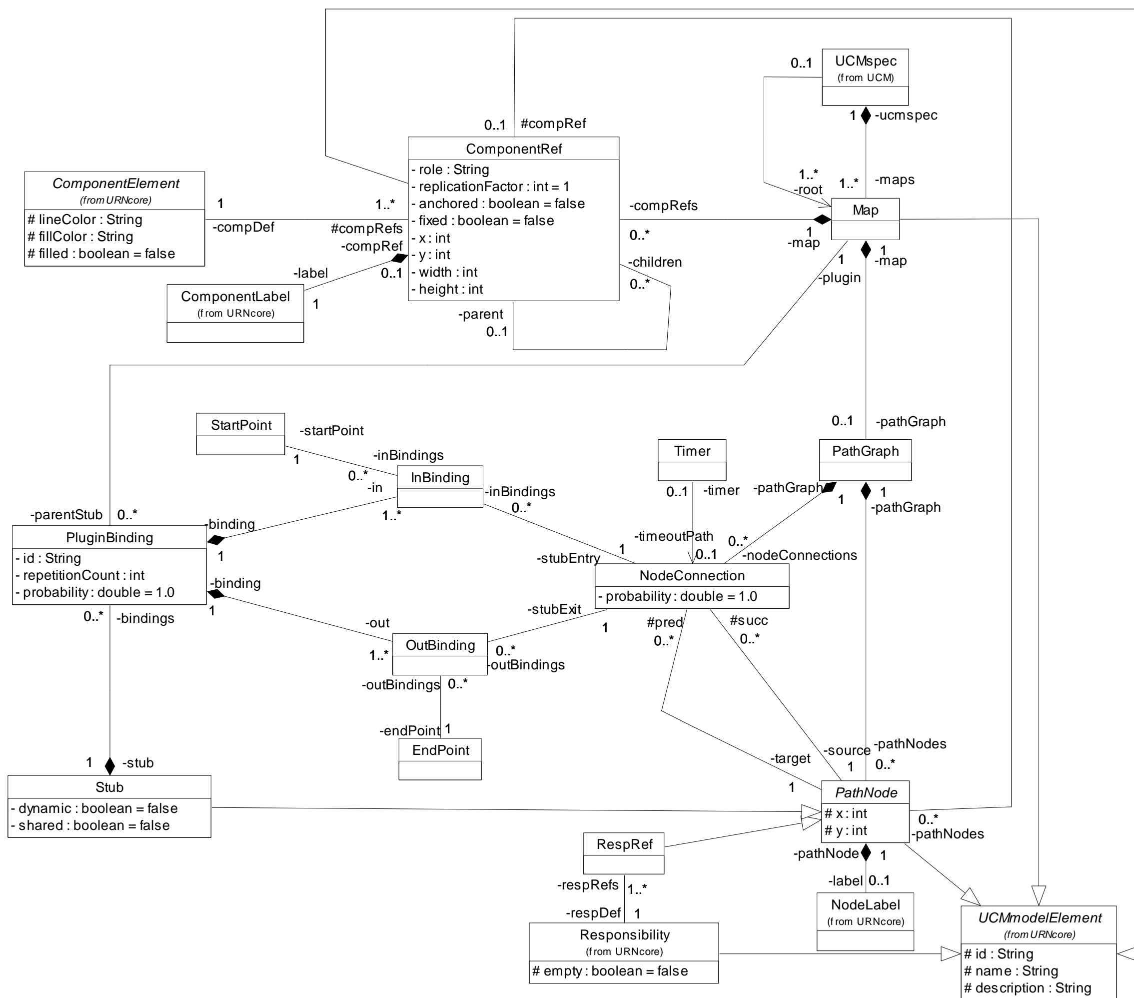

Fig. 6. Vue partielle du paquetage UCM : spécification UCM avec liens entre diagrammes et liens entre nœuds et composantes.

Nous avons effectué des mises à jour au méta-modèle à quelques reprises et ces modifications se propagent à l'outil en quelques minutes, si l'outil est légèrement couplé avec les classes, relations, méthodes ou attributs modifiés. Plus intéressant encore, le développeur peut modifier le code généré et indiquer à EMF de ne pas écraser ses changements lors des mises à jour subséquentes. Le code généré par EMF est très intéressant car il soulage les développeurs de tâches répétitives, comme les notifications et les énumérations, et il encourage l'utilisation de la réflexion sous Java.

EMF implémente toute classe du modèle UML en divisant l'implémentation de l'interface. L'interface générée est une sous-interface d'`EObject`, elle-même sous-classe de `Notifier`. Puisque toute classe implémente `Notifier`, d'autres objets peuvent s'enregistrer pour recevoir des notifications de changements concernant l'objet sous observation. Cette capacité est d'importance capitale lors de la création d'un éditeur car des modifications à l'éditeur principal doivent être reflétées dans les différentes vues et vice-versa. Les éditeurs Eclipse en général, et surtout les éditeurs graphiques faisant usage de GEF, sont fortement basés sur le patron de conception modèle-vue-contrôleur (MVC) [20]. De plus, ces notifications indiquent quels attributs ont été changés, ouvrant la porte à une écoute sélective plus efficace. Enfin, l'implémentation EMF des références bidirectionnelles permet au développeur de modifier un sens de la référence et d'avoir l'inverse effectué automatiquement. Par exemple, étant donné une définition pouvant avoir plusieurs références (association un à plusieurs), si la référence est changée vers une autre définition, l'ancienne définition retirera la référence de sa liste de références et la nouvelle l'ajoutera.

Enfin, puisque toutes les classes implémentent l'interface `EObject`, elles sont dotées de méthodes utilitaires permettant un accès facile à la réflexion. Celles-ci utilisent des classes propres à EMF réalisant la réflexion de façon plus efficace.

L'utilisation d'EMF comme méta-modèle pour l'application jUCMNav permet de réduire les efforts nécessaires pour gérer l'évolution de la notation et des transformations supportées. Ces efforts sont énormément onéreux pour l'actuel UCMNav; les récents ajouts à cette plateforme désuète [15][32] ont été difficilement réalisés et auraient bénéficié d'une meilleure documentation du méta-modèle (externe à l'implémentation) et de l'automatisation offerte par EMF.

IV. INTERFACE DE JUCMNAV BASÉE SUR GEF

A. Aperçu de l'interface utilisateur

L'interface utilisateur de jUCMNav, basée sur Eclipse et GEF, est présentée à la Fig. 7

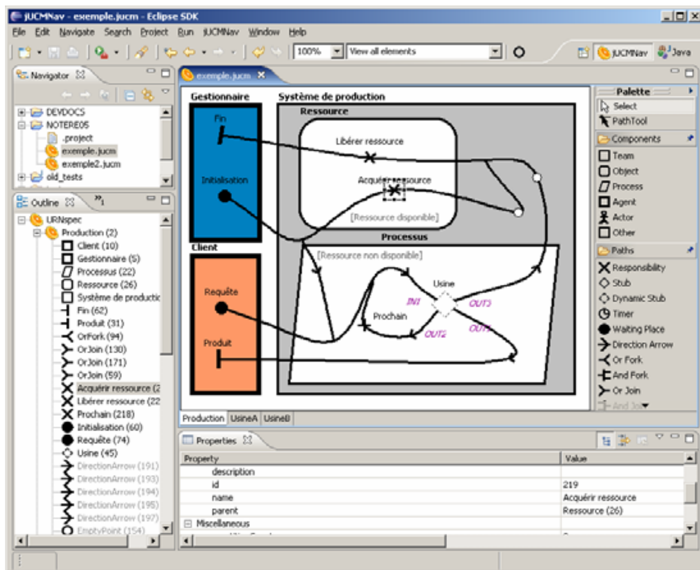

Fig. 7. Interface utilisateur de jUCMNav.

Cette interface est divisée en différents panneaux. L'emplacement par défaut de ces panneaux est déterminé par la perspective fournie par jUCMNav. Les plus importants sont l'éditeur lui-même (au centre), la vue des propriétés (au bas) et la vue de l'aperçu (*Outline*, en bas à gauche). La vue du navigateur de projets, les menus et la première rangée de boutons du haut sont fournis par Eclipse. L'éditeur, contenant la palette d'outils et l'aperçu utilisent la plateforme GEF pour représenter visuellement l'instance du méta-modèle. Une dernière vue située derrière la vue de l'aperçu décrit les responsabilités sur le diagramme courant.

Une fenêtre de dialogue supplémentaire (Fig. 8) a été ajoutée pour permettre de relier des sous-diagrammes UCM à un stub contenu dans un diagramme parent (ce qui correspond aux PluginBinding discutés à la Fig. 6).

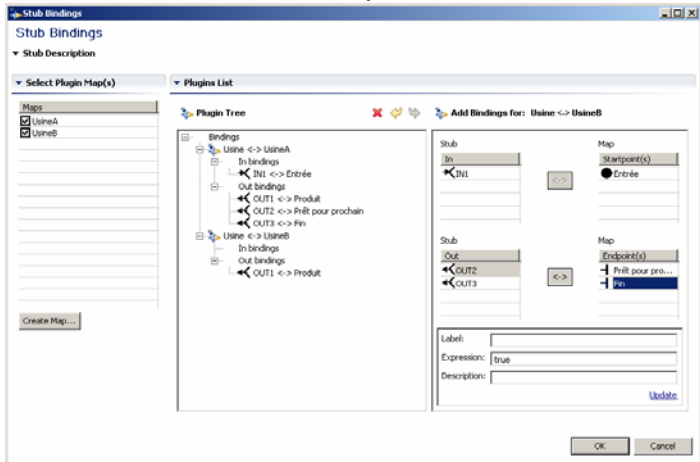

Fig. 8. Dialogue permettant de relier des sous diagrammes UCM à un stub.

Cette fenêtre permet de sélectionner le ou les sous-diagrammes UCM et d'associer leurs points de départ et d'arrivée aux segments d'entrée/sortie du stub (identifiés par IN1, OUT1, …). Puisque l'interface générale dans laquelle s'intègre jUCMNav est fournie par la plateforme Eclipse, l'utilisateur n'a pas à se familiariser avec une interface nouvelle, à moins de ne jamais avoir utilisé Eclipse auparavant. De plus, l'extensibilité étant un point fort de l'architecture d'Eclipse, des additions uniformes sont très facilement créées, bénéficiant autant les développeurs que les utilisateurs de jUCMNav.

B. Utilisation de GEF

L'architecture imposée par GEF lors de la création d'éditeurs graphiques facilite énormément la maintenance et l'amélioration de l'éditeur. En effet, le développement de notre outil a démontré que l'ajout de nouvelles fonctionnalités était souvent très simple, voire trivial, une fois l'apprentissage des cadres de développement Eclipse et GEF complété.

Chaque objet éditable dans GEF (forme, nœud ou arc d'un graphe, etc.) est lié à un objet du modèle EMF. Ceci n'est pas imposé, mais le lien est logique et naturel. Lorsqu'un utilisateur sélectionne un élément du modèle graphique dans l'éditeur principal ou la vue *Outline*, la plateforme Eclipse demande au contrôleur de l'architecture MVC de cet élément (son *EditPart*) pour un *PropertySource*. Le contrôleur génère et retourne donc une nouvelle instance d'un *PropertySource*, une usine à propriétés qu'il a définie au préalable [20]. Les propriétés sont une liste hiérarchique de contrôles, tels des boîtes de texte, des menus déroulants, des dialogues de sélection de couleur, etc., auxquels sont associés un certain attribut ou une référence de l'objet du modèle contrôlé par l'*EditPart*.

jUCMNav vise le développement d'une architecture extensible pour permettre l'implémentation de la notation URN complète. Suivant cet objectif, l'usine de propriétés d'un élément du modèle devient logiquement une énumération des attributs et références de l'objet, obtenues par réflexion. Les types de base ont des contrôles textuels spéciaux, les références vers une instance d'une classe de stéréotype énumération dans le modèle UML est transformé en menu déroulant. De nouveaux attributs supplémentaires (ou des références vers des éléments de modèle pour lesquels les contrôles de propriétés ont déjà étés définis) dans le méta-modèle UCM sont souvent propagés automatiquement du modèle à l'éditeur. Par conséquent, l'intégration des attributs de performance (voir Fig. 3), dont certains sont disponibles dans UCMNav, seront facilement transférables à jUCMNav.

V. FONCTIONNALITÉS DE JUCMNAV

La première version de jUCMNav, rendue publique le 18 juillet 2005 sur http://www.softwareengineering.ca/jucmnav/, offre déjà un bon nombre de fonctionnalités présentes dans UCMNav :

- Support des éléments/nœuds de base (voir palette de la

Fig. 7) et des différents types de composantes;

- Prévention de plusieurs catégories de diagrammes syntaxiquement incorrects;
- Connecteurs courbes (*B-splines*) reliant les nœuds;
- Association d'éléments et de sous-composantes à des composantes parent, et prise en charge lors de déplacements et de redimensionnements;
- Édition par menus contextuels pop-up;
- Associations entre stubs et sous-diagrammes UCM;
- Description des divers éléments;
- Déplacement et effacement de groupe d'éléments;
- Agrandissements (*zoom*), barres de déroulement;
- Sauvegarde/chargement de modèles UCM;
- Exportation des images (`.bmp` et `.jpg` au lieu de `.eps`);
- Disponibilité multiplateforme (via Eclipse) testée sur GNU/Linux, MS Windows™ et Mac OS X™;
- Aide en ligne.

De plus, de nouvelles fonctionnalités tirant avantage d'Eclipse ont été ajoutées :

- Annuler/rétablir (*undo/redo*) à niveaux multiples, grâce au patron de conception *commande* [20] utilisé systématiquement pour toute action dans jUCMNav;
- Placement automatique des éléments (*auto-layout*) basé sur *GraphViz* [19], permettant aussi de gérer des modèles UCM où les coordonnées et les tailles sont absentes, ce qui est utile dans un contexte de rétro-ingénierie et de visualisation de traces [10];
- Conception de diagrammes par glisser-déposer;
- Vue d'appoint hiérarchique (*Outline*) éditable;
- Rétroaction lors de manipulations du diagramme;
- Étiquettes (noms) déplaçables et éditables directement sur le diagramme;
- Interface utilisateur suivant les normes Eclipse et utilisant les services de l'environnement de fenêtrage du système d'exploitation de l'utilisateur;
- Suite de tests fournie avec le code, accompagnée de l'architecture permettant d'exécuter ces tests et ainsi d'assurer une certaine robustesse dans l'application.

VI. RETOUR D'EXPÉRIENCE

Le développement de jUCMNav comme successeur à UCMNav a offert une expérience enrichissante nous ayant permis d'évaluer la viabilité de la plateforme Eclipse comme support d'outils de génie logiciel. Voici un sommaire de quelques difficultés et surprises plaisantes rencontrées.

A. Difficultés rencontrées

- *Courbe d'apprentissage Eclipse* : Eclipse étant une plateforme énorme et sophistiquée, il est inutile de chercher à comprendre tous ses détails en profondeur dès le début. La lecture de quelques chapitres d'un livre d'introduction à Eclipse [10][25] est suffisante pour débuter. Ensuite, les forums de discussion, l'API d'Eclipse et des livres plus techniques tels que [5] offrent des détails plus pertinents.
- *Courbe d'apprentissage GEF* : GEF a une courbe d'apprentissage plus abrupte, et les guides explicatifs n'abondent pas. Nous avons trouvé un excellent guide d'introduction [16] mais nous ne l'avons découvert qu'après avoir surmonté les difficultés initiales grâce à un RedBook d'IBM [18].
- *Les premiers pas* : La première tâche consiste à mettre en place l'architecture générale de l'éditeur, ce qui requiert quelques milliers de lignes de code. Cette phase est ardue: les messages d'erreurs et les exceptions abondent et l'ensemble est souvent non-fonctionnel. Il est aussi difficile de diviser la tâche de créer le noyau du système. Cependant, Eclipse possède maintenant quelques gabarits pour simplifier le travail initial.

Solution : En somme, la lecture seule n'est pas suffisante et il faut absolument mettre la main à la pâte et développer un prototype. Notre conseil est d'assigner à deux programmeurs, programmant en paire, la tâche de refaire le cheminement nécessaire à la création des éditeurs fournis en exemple. Même s'ils font beaucoup de copier-coller, ils auront une bien meilleure compréhension de l'architecture générale et pourront aider les autres par la suite. Suivant ce premier mais modeste accomplissement, un processus itératif et incrémental est facilement mise en place. Les nouvelles fonctionnalités peuvent souvent être développées de façon indépendante, étant donné l'architecture Eclipse et GEF.

- *Sémantique de la notation UCM* : La notation UCM possède des complexités sémantiques peu apparentes à priori. Par exemple, l'effacement d'un point de départ ou d'arrivée dans l'outil efface le plus court segment du scénario possible. Traverser le scénario jusqu'au prochain branchement est simple tout comme le fait de se déconnecter du prochain branchement. La complexité cependant augmente lorsqu'on considère toutes les données implicitement reliées aux multiples éléments effacés et elle explose lorsqu'on doit considérer les boucles implicites. En effet, les boucles implicites sont légales en UCM mais avec plusieurs restrictions.

Solution : Sans entrer dans les détails, les concepts informellement définis ont été transformés en algorithmes plus formels supportant les différents scénarios pouvant être modélisés avec des UCM. Nous avons uniformisé la structure d'effacement pour nous assurer que la même logique s'applique aux effacements explicites et implicites.

- *Tests automatisés* : La mise en place notre processus d'intégration continue n'était pas triviale. Puisque JUnit est intégré dans Eclipse, il est très simple de créer des tests pour des applications et même des plug-ins. Lors du développement d'un plug-in, le développeur exécute une deuxième copie d'Eclipse, celle-ci avec le plug-in d'installé. Ce dernier peut ainsi exécuter ses tests sans af-

fecter l'environnement de développement. Le problème survient lorsqu'on tente d'exécuter ces tests JUnit de la ligne de commande dans des compilations périodiques fournies par des logiciels tels que CruiseControl [7].

Solution : Nous avons eu recours à l'infrastructure de test d'Eclipse lui-même pour pouvoir tester notre plug-in dans notre processus automatisé. L'infrastructure Eclipse installe une copie neuve d'Eclipse sur le disque, y copie le plug-in, débute une nouvelle instance de l'application et exécute les tests. Quelques modifications pour y ajouter l'installation d'une copie neuve de GEF, EMF et jUCMNav et le tour est joué. Malgré le temps requis pour configurer notre processus, le tout en valut la peine avec 450 versions produites en 5 mois.

- *Commandes - construction versus exécution* : Avec GEF, on doit vérifier si un élément est effaçable lors de sa sélection. Si oui, alors l'option d'effacement est activée dans l'éditeur. Lorsqu'on efface plusieurs éléments simultanément, chaque item sélectionné est effacé successivement. Cependant, comme l'effacement d'un élément peut avoir un impact plusieurs autres éléments, l'état du modèle change entre la construction et l'exécution de chacune des commandes. Ceci met en évidence un problème que nous avons rencontré en rapport avec les concepts de *Command* et de *CompoundCommand* en GEF. Ainsi, il est impossible de déterminer si une commande est exécutable au moment de sa construction, comme le voudrait la plateforme. Les algorithmes d'effacement mentionnés précédemment ne peuvent donc pas construire des *CompoundCommands* avant leur exécution.

Solution : Nous avons fait l'hypothèse que nos commandes d'effacement peuvent toujours être exécutées et nous avons uniformisé les commandes utilisées dans l'effacement pour qu'elles définissent la tâche à effectuer seulement lors de son exécution. Ceci a donc requis l'ajout de vérifications additionnelles dans plusieurs commandes.

B. Surprises plaisantes

La programmation sous Eclipse est très agréable; l'éditeur est tout simplement excellent et l'environnement d'exécution nous a agréablement surpris.

- *Génération de code EMF* : La génération du méta-modèle EMF à partir du diagramme de classe fonctionne très bien et le tout est brillamment intégré à GEF.
- *Passage de 3.0.1 à 3.1* : Nous avons développé la version 1.0.0 de jUCMNav pour Eclipse 3.0.1 mais, au moment de sa finalisation, la version 3.1 est apparue. Malgré quelques avertissements mineurs lors de la compilation et lors de l'exécution, le plug-in fonctionnait encore à merveille. Nous avons été informés des changements qui brisaient la compatibilité avec les anciens plugins mais, heureusement pour nous, ils n'affectaient pas jUCMNav.
- *Performance* : Nous n'avons eu qu'un seul problème de performance sérieux, et il était lié à la reconstruction trop fréquente des propriétés lorsque nous déplacions des éléments. Ceci ralentissait parfois l'application de façon visible. Nous avons appris par inadvertance la cause du problème : nous pouvions éviter cette reconstruction coûteuse en réécrivant les méthodes *hashCode()* et *equals()* d'une clé unique utilisée dans nos propriétés. Ne pouvant pas comparer nos entrées, la plateforme recréait le tout, causant ainsi le problème de performance.
- *Peu de couplage* : Les UCM utilisent des courbes comme scénarios, ce qui se différencie des connexions point-à-point utilisées entre les éléments dans GEF. Notre implémentation initiale suivait les concepts GEF, mais une inspection de code décela que le paradigme utilisé n'était pas idéal, puisqu'il redessinait les courbes de façon inefficace. Nous avons donc refait le routeur de connexions, cette fois en se basant sur l'architecture EMF sous-jacente. Les changements furent localisés dans une seule classe et l'impact fut plus que satisfaisant : notre suite de tests s'exécutait 40% plus rapidement.
- *Outils libres* : La qualité et la puissance des outils de développement gratuits d'aujourd'hui est épatante. En plus des Eclipse, GEF, EMF et JUnit mentionnés précédemment, nous avons utilisé TWiki [27] pour la gestion des exigences et la documentation, BugZilla pour les bogues, Ant et CruiseControl [7] pour l'intégration continue et StatCVS pour suivre l'évolution du projet via notre système de contrôle de versions (CVS). Plusieurs autres outils libres eurent un impact bénéfique sur notre projet, nous permettant un développement plus structuré mais surtout plus simple qu'il y a 8 ans, lors des premiers efforts de développement de UCMNav.
- *Développement itératif* : La simplicité de l'ajout de plusieurs nouvelles fonctionnalités est surprenante. Les modifications sont souvent très localisées et des améliorations puissantes peuvent être faites rapidement.
- *Généralisations* : Tel que mentionné précédemment, notre implémentation de la vue des propriétés utilise la réflexion pour afficher les attributs et associations des objets du modèle. À l'aide d'une dizaine de lignes de code, notre application ajouta des menus déroulants pour chaque énumération définie dans le méta-modèle. Nous songeons à généraliser davantage ces capacités pour inférer le type d'éditeur non pas simplement par le type de données et le nom de l'attribut mais par des informations additionnelles indiquées dans le méta-modèle. Il serait plausible d'extraire les informations sur les catégories et les groupements hiérarchiques et imbriqués du méta-modèle par un processus similaire. Avec un minimum d'effort, la vue des propriétés seraient complètement déterminée par le méta-modèle et très peu de code additionnel serait ainsi nécessaire.

VII. Conclusion

Cet article a présenté jUCMNav, une nouvelle plateforme pour le développement de modèles Use Case Map. Ce nouvel outil cherche à combler les lacunes de l'outil actuel (UCM-Nav) en proposant une architecture évolutive et robuste (basée sur Eclipse, EMF et GEF) et en offrant une convivialité élevée et standard, tout en conservant la portabilité de l'application et en éliminant la dépendance aux serveurs *X Window*.

L'outil jUCMNav est encore en cours de développement. Bien que les fonctionnalités de l'outil reliées à l'édition de modèles UCM soient presque terminées, la partie analyse reste encore à implémenter. Certaines fonctionnalités sont planifiées à court et moyen terme :

- Support pour responsabilités dynamiques;
- Distinction entre classes et instances de composantes;
- Lecture/écriture du format de fichier UCMNav;
- Lecture/écriture du format de fichier URN (XMI);
- Recherche dans un modèle;
- Support des définitions de scénarios, traversée du modèle UCM et génération de scénarios individuels (avec sérialisation XML), selon [2];
- Intégration avec Telelogic DOORS via des scripts et une librairie DXL, tel que décrit dans [15];
- Génération de rapports (en HTML et/ou PDF);
- Support des annotations de performance et génération de modèles CSM, selon [32];
- Support de la notation GRL (partie de URN [1]) et de liens de traçabilité entre GRL et UCM.
- Création de points d'extensions Eclipse afin que d'autres plug-ins puissent étendre jUCMNav.

jUCMNav permettra ainsi de modéliser et d'analyser plus facilement, à l'aide de la notation UCM, plusieurs types de systèmes réactifs, concurrents et répartis complexes, et ce en collaboration avec d'autres outils et notations.

Nous espérons aussi pouvoir utiliser cette nouvelle plateforme pour étudier de nouveaux concepts tels que les « aspects » appliqués aux modèles orientés scénarios [14].

Remerciements

Les auteurs remercient Jean-François Roy, Gunter Mussbacher, Jacques Sincennes et Ali Echihabi pour leurs conseils et commentaires lors du développement de jUCMNav.

Références

[1] D. Amyot, "Introduction to the User Requirements Notation: Learning by Example". *Computer Networks*, 42(3), 285-301, 21 June 2003.

[2] D. Amyot, D.Y. Cho, X. He et Y. He, "Generating Scenarios from Use Case Map Specifications". *Third International Conference on Quality Software (QSIC'03)*, Dallas, É-U, novembre 2003, 108-115.

[3] D. Amyot et L. Logrippo, "Use Case Maps and LOTOS for the Prototyping and Validation of a Mobile Group Call System". *Computer Communication*, 23(12), 2000, 1135-1157.

[4] D. Amyot, J.-F. Roy et M. Weiss, "UCM-Driven Testing of Web Applications". *12th SDL Forum (SDL 2005)*, Grimstad, Norvège, juin 2005. LNCS 3530, Springer, 247-264.

[5] J. Arthorne et C. Laffra, *Official Eclipse 3.0 FAQs*, Addison-Wesley, É-U, 2004.

[6] R.J.A Buhr et R.S. Casselman, *Use Case Maps for Object-Oriented Systems*, Prentice-Hall, É-U, 1995.

[7] CruiseControl project, *CruiseControl*. Disponible à http://cruisecontrol.sourceforge.net

[8] Eclipse project, *Graphical Editing Framework (GEF)*. Disponible à http://www.eclipse.org/gef/

[9] Eclipse project, *Eclipse Modeling Framework (EMF)*. Disponible à http://www.eclipse.org/emf/

[10] E. Gamma et K. Beck, *Contributing to Eclipse – Principles, Patterns, and Plug-Ins*, Addison-Wesley, É-U, 2004.

[11] A. Hamou-Lhadj, E. Braun, E., D. Amyot et T. Lethbridge, "Recovering Behavioral Design Models from Execution Traces". *9th European Conference on Software Maintenance and Reengineering (CSMR)*, Manchester, UK, mars 2005. IEEE Computer Society, 112-121.

[12] IBM, *Rational Rose Enterprise Edition 2003*. http://www-306.ibm.com/software/awdtools/developer/rose/

[13] IBM, *Rational Software Architect*, 2005. http://www-128.ibm.com/developerworks/rational/products/rsa/

[14] I. Jacobson et P.-W. Ng, *Aspect-Oriented Software Development with Use Cases*, Addison-Wesley, 2005.

[15] B. Jiang, *Combining Scenarios with a Requirements Management System*. M.Sc. thesis, University of Ottawa, juin 2005.

[16] R. Lemaigre, *GEF Description,* Disponible à http://eclipsewiki.editme.com/GefDescription

[17] A. Miga, *Application of Use Case Maps to System Design with Tool Support*. M.Eng. thesis, Dept. of Systems and Computer Engineering, Carleton University, Ottawa. October 1998. Disponible à http://www.UseCaseMaps.org/tools/ucmnav/

[18] W. Moore, D. Dean, A. Gerber, G. Wagenknecht et P. Vanderheyden, *Eclipse Development using the Graphical Editing Framework and the Eclipse Modeling Framework*. IBM Red Book, 2004. Disponible à http://www.redbooks.ibm.com/abstracts/sg246302.html

[19] S. North et al., *Graphviz*, 2005. Disponible à http://www.graphviz.org/

[20] Oberlin College, *Design Patterns*, 2005. Disponible à http://exciton.cs.oberlin.edu/JavaResources/DesignPatterns/

[21] OMG, *UML 2.0 Superstructure Specification*, OMG Adopted Specification, April 30, 2004. Disponible à http://www.uml.org/#UML2.0

[22] OMG, *XML Metadata Interchange (XMI)*, mai 2005, version 2.0. Disponible à http://www.omg.org/docs/formal/05-05-01.pdf

[23] D.B. Petriu, D. Amyot et M. Woodside, "Scenario-Based Performance Engineering with UCMNav". *11th SDL Forum (SDL'03)*, Stuttgart, Allemagne, juillet 2003. LNCS 2708, 18-35.

[24] D.B. Petriu et M. Woodside, "An Intermediate Metamodel with Scenarios and Resources, for Generating Performance Models from UML Designs". À paraître dans *Software and Systems Modeling*.

[25] S. Shavor, J. D'Anjou, S. Fairbrother, D. Kehn, J. Kellerman et P. McCarthy, *The Java Developer's Guide to Eclipse*, Addison-Wesley, É-U, 2003.

[26] Telelogic AB, *DOORS/ERS*. http://www.telelogic.com/products/doorsers/

[27] TWiki.org, *TWiki*, *Disponible à http://www.twiki.org*

[28] Union Internationale des Télécommunications, *Recommendation Z.150 (02/03), User Requirements Notation (URN) – Language Requirements and Framework*. Genève, Suisse, 2003.

[29] Union Internationale des Télécommunications, Commission d'études 17, "New draft metamodel for URN". Document temporaire SG17-050330-TD-WP3-3044, avril 2005.

[30] M. Weiss et D. Amyot, "Business Process Modeling with URN". *International Journal of E-Business Research*, 1(3), 63-90, July-September 2005.

[31] X.Org Foundation, *X Window System*, 2005. http://www.x.org/

[32] Y.X. Zeng, *Transforming Use Case Maps to the Core Scenario Model Representation*. M.Sc. thesis, University of Ottawa, juin 2005.

[33] T.C. Zhao et M Ovenmars, *XForms, a GUI toolkit for X*, 2004. Disponible à http://savannah.nongnu.org/projects/xforms