Validation d'interprètes OCL

Myriam Ballarin

CONSERVATOIRE NATIONAL DES ARTS ET METIERS
CENTRE REGIONAL DE TOULOUSE

MEMOIRE

Présenté en vue d'obtenir

LE DIPLOME D'INGENIEUR CNAM

en

INFORMATIQUE
SYSTEMES D'INFORMATION

Par

Myriam BALLARIN

Lieu du stage : Institut de Recherche en Informatique de Toulouse (IRIT)
Equipe MACAO
Responsable en entreprise : Monsieur Thierry MILLAN

Validation d'interprètes OCL

Soutenu le : mai 2012

JURY

PRESIDENT :	**M. Yann Pollet – CNAM Paris**
MEMBRES :	**M. Hervé Leblanc –Université Paul Sabatier**
	M. Christian Percebois - Université Paul Sabatier
	M. Thierry Millan - Université Paul Sabatier
	M. Hadj Batatia - CNAM

REMERCIEMENTS

Je tiens à remercier tous les membres de l'équipes MACAO de l'IRIT et tout spécialement Monsieur Thierry Millan pour sa disponibilité, sa bonne humeur, ses qualités pédagogiques et son soutien qui m'ont permis de tirer le meilleur parti de mon stage et notamment de progresser en expérimentant les connaissances acquises au cours de mes études au CNAM.
Je tiens aussi à remercier Monsieur Christian Percebois pour son accueil chaleureux et son attention tout au long de mon stage.
Merci aussi à Monsieur Jean-Paul Bodeveix de l'IRIT, Monsieur Pierre Gaufillet d'AIRBUS et Monsieur Sébastien Gabel de C-S pour leur aide précieuse à la résolution du problème de lancement des requêtes OCL en batch, pour mener à bien mon étude et pour maitriser TOPCASED en ce qui concerne la partie OCL.
Merci à ma mère qui a été d'un grand soutien ainsi qu'à mon compagnon, ma famille, et mes amies Laure, Marie-Christine, Iulia.

SOMMAIRE

LISTE DES ABREVIATIONS

API	Application Program Interface
CRUD	« Create, Read, Update, Delete » pour la mise à jour d'une base de données
DAO	Data Access Object est un objet d'accès aux données, c'est un patron de conception
DSL	Domain specific langage
DSL	Domain specific langage
DSML	Directory Service Markup Langage
DTD	Data type définition
IDE	environnement de développement intégré
MDE	: Model Driven Engineering : Ingénierie dirigée vers les modèles
MOF	Meta Object Facility
OCL	Object Constraint Langage
OMG	Object Management Group (organisation mondiale qui définit les normes)
PLPM	Production logicielle pilotée par les modèles
SAX	parseur de JAVA
XMI	XML Metadata Interchange
XML	*eXtensible Markup Language*

AVANT PROPOS SUR L'IDM, et LE MDA.

Cet avant-propos à pour but de définir, d'introduire les concepts centraux pour la compréhension de ce rapport.

1 L'IDM, INGÉNIERIE DIRIGÉE PAR LES MODÈLES

Les citations, les idées et les concepts énoncés dans ce chapitre proviennent de l'article «Ingénierie dirigée par les modèles (IDM), état de l'art » de Combemale (2009) [Com09].

L'IDM (Ingénierie Dirigée par les Modèles) possède comme concept de base le modèle. Le modèle est considéré comme une abstraction d'un système réel, vue sous un certain angle. Pour décrire ces modèles, l'IDM introduit la notion de méta-modèle qui définit le langage de modélisation. En cela, « *l'IDM favorise la définition de langages de modélisation dédiés à un domaine particulier (Domain Specific Modeling Language – DSML) offrant ainsi aux utilisateurs des concepts propres à leur métier et dont ils ont la maîtrise.* ».

Devant la diversité les méta-modèles possibles et afin de donner un cadre général pour la description de ces mêmes méta-modèles, l'OMG a pris l'initiative de proposer un langage de définition sous la forme d'un « méta-méta-modèle » MOF (Meta Object Facility). Ce « méta-méta-modèle » a la propriété de se décrire lui-même (méta circularité). Le « méta-méta-modèle », le « méta-modèle », le modèle et le système réel peuvent être représentés graphiquement par la pyramide de modélisation de l'OMG qui reflète les niveaux d'abstractions associés à chacun de ces niveaux. À la base de cette pyramide, le monde réel, le concret (couche M0). En haut, le pyramidion qui représente l'abstraction la plus élevée, le « méta-méta-modèle » (couche M3) (cf. Figure 1). Entre les deux, la couche M2 (le « méta-modèle ») qui décrit la grammaire utilisée pour le langage de modélisation de la couche M1, la couche modèle.

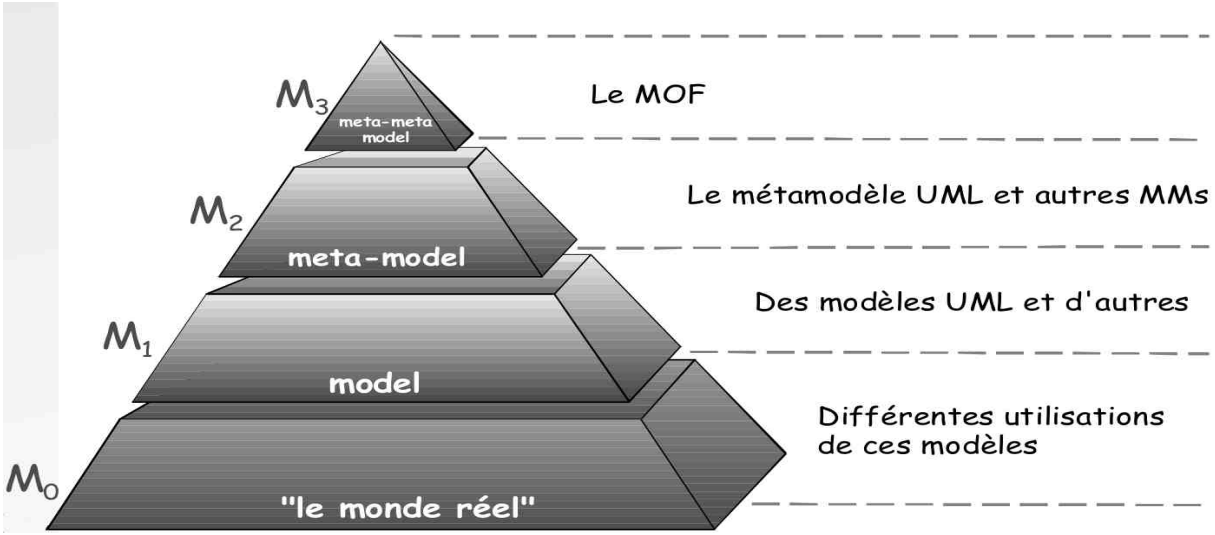


Fig. 1: Pyramide de modélisation de l'OMG (Jean Bévizin)

D'après l'article du site

http://www.megaplanet.org/jean-marie-favre/papers/IDMIHMUnTandemPrometteur.pdf

En IDM, « tout est système » ; les notions de modèles et de méta-modèles ne sont que relatives. Ainsi, par exemple, un système particulier comme une mappemonde ne joue le rôle de modèle que par rapport à un système, la planète Terre en l'occurrence. Si l'on ne considère pas ce lien, la mappemonde n'est qu'un système. La mappemonde ne jouera le rôle de modèle pour la Terre que pour une personne établissant ce lien de représentation entre les deux instances. Les notions de rôles et de relations sont prédominantes en IDM. L'IDM s'articule autour de trois concepts et trois relations :

• Les modèles et la relation « Représente ». Il n'existe pas de définition universelle du concept de modèle, cependant un consensus admet que modèle et système étudié sont deux rôles complémentaires. Un modèle représente un système.

• Les méta-modèles et la relation « EstConformeA ». Un méta-modèle est un modèle d'un langage de modélisation. Pour être traitable par une machine, un modèle doit être conforme à un méta-modèle. Par exemple, une phrase est conforme (ou non) à une grammaire.

• Les transformations et la relation « EstTransforméEn ». Cette relation est une mise en correspondance entre éléments d'un ou plusieurs modèles. Le cœur de l'IDM consiste à représenter de manière explicite les transformations, donnant lieu à la notion de modèle de transformation. Ces modèles se basent sur les méta-modèles des modèles que l'on désire transformer. L'objectif est double : capitaliser un savoir-faire méthodologique et envisager l'automatisation.

Dans le contexte de l'IDM, la notion de transformation de modèle joue un rôle fondamental, aussi de nombreux outils, tant commerciaux que dans le monde de l'open source sont aujourd'hui disponibles pour faire la transformation des modèles. On peut grossièrement distinguer quatre catégories d'outils :

- Les outils XML et de transformation de graphes par exemple XSLT (extensible Stylesheet Language for Transformations) ou Xquery.

  XSLT fournit un langage extensible et puissant pour transformer les documents XML soit en HTML, en PDF, ou en code JAVA par exemple.

  D'après le site :

  http://fr.wikipedia.org/wiki/Ing%C3%A9nierie_dirig%C3%A9e_par_les_mod%C3%A8les#Cat.C3.A9gorie_des_outils_XML_et_des_outils_de_transformation_de_graphes

  L'expérience montre que ce type de langage est assez mal adapté pour des transformations de modèles complexes (c'est-à-dire allant au-delà des problématiques de transcodage syntaxique), car ils ne permettent pas de travailler au niveau de la sémantique (c'est-à-dire l'étude de la signification du code (le fond) par opposition à l'analyse syntaxique qui étudie la forme c'est-à-dire la façon dont le code est agencé) des modèles manipulés mais simplement à celui d'un arbre couvrant le graphe de la syntaxe abstraite du modèle ce qui impose de nombreuses contorsions qui rendent rapidement ce type de transformation de modèles complexes à élaborer, à valider et surtout à maintenir sur de longues périodes.

- Les outils de transformation des modèles via des AGL commerciaux

  D'après le site :
  http://fr.wikipedia.org/wiki/Ing%C3%A9nierie_dirig%C3%A9e_par_les_mod%C3%A8les#Cat.C3.A9gorie_des_outils_XML_et_des_outils_de_transformation_de_graphes.
  L'intérêt de cette catégorie d'outils de transformation de modèles est d'une part leur relative maturité et d'autre part leur excellente intégration dans l'atelier de génie logiciel qui les héberge. Leur principal inconvénient est le revers de la médaille de cette intégration poussée : il s'agit la plupart du temps de langages de transformation de modèles propriétaires sur lesquels il peut être risqué de miser sur le long terme. De plus, historiquement ces langages de transformation de modèles ont été conçus comme

des ajouts au départ marginaux à l'atelier de génie logiciel qui les héberge. Même s'ils ont aujourd'hui pris de l'importance, ils ne sont toujours pas vus comme les outils centraux qu'ils devraient être dans une véritable ingénierie dirigée par les modèles. De nouveau ces langages montrent leur limitation lorsque les transformations de modèles deviennent complexes et qu'il faut les gérer, les faire évoluer et les maintenir sur de longues périodes. En voici deux exemples :

- TROPIC =*A Framework for Model Transformations on Petri Nets in Color* (voir : http://www.modeltransformation.net/)
- ATL: « A Transformation Language » fournit des façons de produire un jeu de modèles cibles à partir d'un jeu de modèles source. Développé en plus de la plate-forme d'Éclipse, l'ATL Environnement intégré (IDE) fournit un certain nombre d'outils de développement standard (l'accentuation de syntaxe, le programme de mise au point, etc.) qui aspire à faciliter le développement de transformations ATL. (voir : http://eclipse.org/atl/).

- La troisième catégorie regroupe les outils conçus spécifiquement pour faire de la transformation de modèles et prévus pour être plus ou moins intégrables dans les environnements de développement standards. Ces langages spécifiquement conçus pour la transformation de modèle ont l'avantage de permettre d'exprimer très simplement les transformations de modèles simples (comme par exemple des transcodages syntaxiques), mais ils trouvent rapidement leurs limites lorsque les transformations de modèles deviennent complexes du point de vue algorithmique, ou bien lorsqu'il faut gérer de nombreuses variantes (contexte des lignes de produits) et les faire évoluer et les maintenir sur de longues périodes.

- La dernière catégorie d'outils de transformation de modèles est celle des outils de méta-modélisation. La transformation de modèles revient alors à l'exécution d'un méta-programme orienté objet, pour lequel il est relativement aisé de construire un Framework facilitant la gestion et la maintenance de nombreuses variantes de transformations nécessaires au contexte des lignes de produits.

La transformation des modèles inclut les étapes de la compilation : voir le site http://fr.wikipedia.org/wiki/Compilateur

- le prétraitement, nécessaire pour certains langages comme C, qui prend en charge la substitution de macro et de la compilation conditionnelle.

  Généralement, la phase de prétraitement se produit avant l'analyse syntaxique ou sémantique ; par exemple dans le cas de C, le préprocesseur manipule les symboles lexicaux plutôt que des formes syntaxiques.

- l'analyse lexicale, qui découpe le code source en petits morceaux appelés jetons (tokens).

  Chaque jeton est une unité atomique unique de la langue (unités lexicales ou lexèmes), par exemple un mot-clé, un identifiant ou un symbole. La syntaxe de jeton est généralement un langage régulier, donc un automate à états finis construits sur une expression régulière peut être utilisé pour le reconnaître.
  Cette phase est aussi appelée le balayage ou lexing ; le logiciel qui effectue une analyse lexicale est appelé un analyseur lexical ou un scanner. Deux exemples classiques : lex et flex.

- **l'analyse syntaxique** implique l'analyse de la séquence jeton pour identifier la structure syntaxique du programme.

  Cette phase s'appuie généralement sur la construction d'un arbre d'analyse ; on remplace la séquence linéaire des jetons par une structure en arbre construit selon la grammaire formelle qui définit la syntaxe du langage. Par exemple, une condition est toujours suivie d'un test logique (égalité, comparaison...). L'arbre d'analyse est souvent modifié et amélioré au fur et à mesure de la compilation. Yacc et GNU Bison sont les analyseurs syntaxiques les plus utilisés.

- l'analyse sémantique est la phase durant laquelle le compilateur ajoute des informations sémantiques à l'arbre d'analyse et construit la table des symboles.

  Cette phase vérifie le type (vérification des erreurs de type), ou l'objet de liaison (associant variables et références de fonction avec leurs définitions), ou une tâche

définie (toutes les variables locales doivent être initialisées avant utilisation), peut émettre des avertissements, ou rejeter des programmes incorrects.

L'analyse sémantique nécessite habituellement un arbre d'analyse complet, ce qui signifie que cette phase fait suite à la phase d'analyse, et précède logiquement la phase de génération de code ; mais il est possible de replier ces phases en une seule passe.

- la transformation du code source en code intermédiaire ;
- l'application de techniques d'optimisation sur le code intermédiaire : c'est-à-dire rendre le programme « meilleur » selon son usage.
- la génération de code avec l'allocation de registres et la traduction du code intermédiaire en code objet, avec éventuellement l'insertion de données de débogage et d'analyse de l'exécution ;
- et finalement l'édition des liens.

Il est important de comprendre cette phase de compilation car nous aurons à étudier le langage OCL dans différents outils, et à interpréter des règles. En effet nous étudierons les résultats de l'exécution de requêtes OCL, tant au niveau de la syntaxe des requêtes grâce à l'interprétation que du résultat attendu lors de l'exécution.

OCL est le langage associé à UML et au MOF afin de de vérifier la sémantique statique lors de la conception des modèles ou des méta-modèles car ils doivent obéir à certaines bonnes pratiques de modélisation. C'est un élément essentiel du MDA.

2 LE MDA, MODEL DRIVEN ARCHITECTURE

En 2000, l'OMG a défini le MDA (Model Driven Architecture) afin de diffuser et promouvoir de bonnes pratiques de modélisation. Le MDA s'appuie sur le standards MOF pour la couche « M3 », sur les standards UML (Unified Modeling Language), SysML (Systems Modeling Language) ou encore CWM (Common Warehouse Metamodel) et bien d'autres « méta-modèles» définis à l'aide du « MOF » pour les couches M2 et M1, et sur le standard d'enregistrement XMI (XML Metadata Interchange). Ce dernier standard « est un format basé sur XML pour les modèles exprimés à partir d'un méta-modèle MOF ». Avec la spécification des standards appliqués aux couches M3, M2 et M1, le MDA peut être considéré comme un sous-ensemble de l'IDM ou encore « une instance particulière de l'IDM ». Le MDA propose de rendre pérenne le modèle d'analyse et de conception en le concevant de manière indépendante de la plate-forme d'exécution (PIM – Plateform Independent Model). Le MDA

considère le code comme un modèle spécifique à la plate-forme (PSM – Plateform Specific Model). Il peut être représenté par les langages C++, Java ou autres. *Figure 2: Le processus en "Y" de l'approche MDD.*

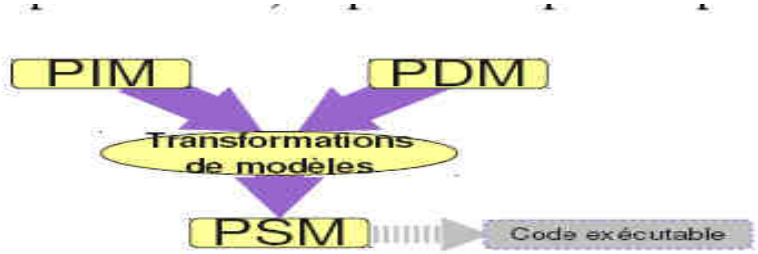


Fig. 2 : Le processus en « Y » de l'approche MDD

Le passage du modèle PIM vers PSM fait intervenir un modèle de description de plate-forme (PDM – Platform Description Model). Par rapport à l'exemple cité plus haut, ce PDM correspond au méta-modèle du langage C++, Java ou autres. L'intervention du PIM et du PDM vers le PSM peut être représentée par une démarche en « Y » (Figure 2). La compilation du code en un exécutable ne fait pas partie directement de cette démarche mais en est la suite logique.

Comme nous l'avons indiqué précédemment, le MDA s'appuie sur des modèles et des méta-modèles qui doivent être bien formés. Afin de spécifier les contraintes nécessaires à la formalisation de règles de bonne formation, un langage d'expression de contraintes a été développé : le langage OCL que nous présentons ci-après dans les concepts.

Une fois les fondements de mon travail posés et avant de présenter mon travail proprement dit, je vais rapidement vous présenter la structure qui m'a accueilli durant mon stage.

L'ORGANISME D'ACCUEIL

Mon organisme d'accueil est l'IRIT (Institut de Recherche en Informatique de Toulouse) situé à l'Université Paul Sabatier de Toulouse

1 LE CENTRE DE RECHERCHE IRIT

L'IRIT est né de la volonté de regrouper différents laboratoires et groupes de recherche en informatique de Toulouse, hébergés initialement à l'Institut National Polytechnique de Toulouse (INPT), à l'Université Paul Sabatier (UPS), à l'Université des Sciences Sociales Toulouse 1 (UT1) et à l'Université du Mirail (UTM). Cet institut fédère les travaux de recherche toulousains en informatique et devient une unité mixte de recherche UMR 5505. Ne connaissant pas le fonctionnement d'un institut de recherche et étant curieuse de son fonctionnement, je décris dans les paragraphes suivants quelques aspects : administratif, les centres de décision, les thèmes de recherches de cet organisme. Une attention particulière est portée sur l'équipe MACAO qui m'accueille : ses centres d'intérêts, ses relations avec le monde industriel et ses perspectives.

1.1 ASPECTS ADMINISTRATIFS

Le financement

À la date de rédaction de ce rapport, les chiffres du financement sont ceux de fin 2008, publiés sur le site de l'IRIT. Le budget global s'élève à 27 307K€ HT.
Il est ventilé comme suit :

- Dotation annuelle : 598 (2,19%),
- Ressources contractuelles : 3 789 (13,88%),
- Salaires : 22 920 (83,93%).

Les dotations et les salaires sont respectivement alloués par le ministère de l'éducation nationale pour les enseignants-chercheurs, par le CNRS pour les chercheurs.
Les ressources contractuelles proviennent de contrats passés avec les acteurs publics et privés pour un montant de 3 789K€ HT. Ces ressources servent à financer les équipements et le fonctionnement du laboratoire.
De plus elles permettent, lorsque c'est possible, de financer le salaire des contractuels du laboratoire. Ces contrats sont passés avec le Ministère et :

- Des organismes publics (1 202K€ 31,72%),
- La région Midi-Pyrénées (737K€ 19,45%),
- Des sociétés industrielles (652K€ 17,21%),
- Les instances européennes (654 K € 17,26%)
- CPER (Contrat Projet État Région) (544K€ 14,36%).

Le personnel

Contrairement aux chiffres du financement, le nombre de personnes présentes à l'IRIT est constamment mis à jour. Six cent vingt-sept personnes travaillent actuellement au sein de cet institut. Elles sont réparties de la manière suivante :

- Enseignants chercheurs (215 – 34,29%),
- Chercheurs (28 – 4,47%),
- ITAITAOS (45 – 7,18%),
- Doctorants (228 – 36,36%).

Le reste de la répartition est formé par les catégories post-doctorant, invités et contractuels (111 – 17,70%). Il est à noter que 92,82% des personnes font de la recherche.

1.2 LE CONSEIL DE LABORATOIRE

Le conseil de laboratoire peut s'assimiler au conseil de direction d'une entreprise. Ce conseil se réunit une fois par mois et prend toutes les décisions relatives à la vie de l'institut. Il est présidé par le directeur de l'IRIT, accompagné par les élus de tous les corps (chercheurs, enseignants-chercheurs, administratifs). Les décisions concernent la gestion des contrats, la politique de rayonnement à l'étranger (accueil ou envoi d'étudiant), la gestion des locaux, la préparation de l'évaluation quadriennale et bien autres aspects de la vie courante.

1.3 LE CONSEIL SCIENTIFIQUE

Le conseil scientifique est un organe consultatif. Il est également présidé par le directeur de l'IRIT, accompagné par les responsables des équipes de recherches. Il traite de toutes les questions relatives à la politique scientifique de l'IRIT. En complément des thèmes scientifiques, ce conseil a défini des axes stratégiques afin de conduire des actions concertées avec les acteurs sociaux économiques et de répondre ainsi à des défis technologiques. Ces

axes se veulent fédérateurs des équipes de recherches : « Informatique pour la santé », « Masses de données et calcul », « Systèmes sociotechniques ambiants » et « Systèmes embarqués critiques ».

1.4 LES RECHERCHES

L'IRIT est organisé autour des 7 thèmes de recherche et chacun de ces thèmes participent de manière diverse aux quatre axes de recherches fédératrices. La liste de ces thèmes, présentés ici, est extraite du site Web de l'IRIT (Site WebIRIT) :

Tableau I: Les 7 thèmes de recherches de l'IRIT.

thème 1	Analyse et synthèse de l'information
thème 2	Indexation et recherche d'informations
thème 3	Interaction, autonomie, dialogue et coopération
thème 4	Raisonnement et décision
thème 5	Modélisation, algorithmes et calcul haute performance
thème 6	Architecture, systèmes et réseaux
thème 7	Sûreté de développement du logiciel

Le sujet de stage s'inscrit dans le thème 7 « Sûreté de développement du logiciel » qui est animé par trois équipes : l'équipe ACADIE, l'équipe ICS et l'équipe MACAO. C'est au sein de cette dernière équipe que je réalise ce stage. Le chapitre suivant détaille l'organisation et les thèmes de recherches de cette équipe MACAO.

2 ÉQUIPE MACAO

2.1 PRÉSENTATION

L'équipe MACAO (Modèles, Aspects, Composants pour des Architectures à Objets), dirigée à ce jour par M. Bernard Coulette, comprend actuellement 12 permanents et 17 membres non permanents dont 13 doctorants, 4 ATER (Attaché Temporaire d'enseignement et de Recherches). Comme dans beaucoup d'équipes de recherche, la diversité géographique mondiale des membres en fait une des richesses des plus précieuses quant au partage des modes de pensées, à l'originalité des approches. L'équipe MACAO n'échappe pas à cette règle

et les thèmes de recherches profitent de cette vitalité. Les pays représentés sont en ce moment : le Liban, le Vietnam, l'Algérie, le Maroc, la Russie et la Roumanie.

2.2 THÈMES DE RECHERCHES

Les axes de recherches sont principalement centrés sur les modèles et l'amélioration de la sureté de développement des logiciels. Comment arriver à prendre en compte l'évolution des modèles au cours du temps, au cours des étapes d'affinement, tout en assurant une cohésion d'ensemble et la traçabilité de ces changements ? Ces questions tendent à être résolues par les personnes impliquées par l'axe de recherche « processus de développement à base de modèles ».

Le thème « ingénierie système et les composants », quant à lui, essaie de répondre aux défis de la complexité grandissante des logiciels devant fonctionner sur des systèmes embarqués à haute sûreté de fonctionnement. La réponse est proposée sous la forme de composants logiciels interopérables. Elle permet de séparer la partie métier de la gestion des ressources contraintes sur les systèmes embarqués.

Que ce soient des composants ou des modèles, le concepteur doit passer par une phase de mise en œuvre de ses modèles en décrivant leurs comportements à l'aide d'un langage de programmation tel que le C, C++, Java ou autres. « L'interopérabilité des modèles et des langages » propose d'« intégrer entre les phases de conception et de génération du code, un niveau de modélisation basé sur les langages de programmation et sur leurs propriétés » (Site Web-IRIT).

Dans l'IDM et particulièrement dans le MDA, l'approche en « Y » implique de fusionner le modèle indépendant de la plate-forme (PIM) avec le modèle spécifique à cette même plate-forme (PSM). D'une manière ou d'une autre, cette opération revient à faire une transformation multi-modèle vers un modèle. L'axe de recherche « transformation de modèles » cherche à formaliser la transformation à l'aide d'une grammaire de graphe attribuée et à exploiter l'interprète pOCL pour réaliser des transformations ou réaliser des animations de modèles.

Le logiciel NEPTUNE offre des fonctionnalités pour manipuler des modèles et méta-modèles à l'aide du langage pOCL. La première version de ce logiciel a été financée par un contrat européen : 5ème plan coordonné de recherches et développements. Une description, des derniers contrats réalisés, en cours ou futur, est donnée ci-après.

2.3 LES CONTRATS, LES ASPECTS FINANCIERS

Les équipes, au sein de l'IRIT, peuvent être vues d'une certaine manière comme des entreprises. Bien que les salaires et la dotation d'équipement soient financés par l'État, pour l'équipe MACAO, cette dotation permet d'assurer le quotidien. Elle ne permet pas d'investir dans des équipements au-delà de l'ordinaire. Elle a besoin de ressources supplémentaires pour s'équiper, pour se développer, accueillir des personnes étrangères, financer les doctorants, les soutenances de thèses et bien d'autres imprévus.

Pour obtenir ces crédits, MACAO participe à des contrats de type européen, national, à la demande d'entreprises ou en collaboration avec celles-ci. En fait, la difficulté vient du fait qu'il s'agit de rechercher l'équilibre dynamique entre des sollicitations R&D à durées limitées et une recherche fondamentale avec des objectifs à longs termes.

Le tableau récapitulatif des contrats illustre les sources de financement. Au regard de la dotation récurrente du laboratoire pour ses équipes, de l'ordre de 400 € en moyenne par chercheur permanent et par an, les contrats R&D MACAO ont représentés, sur les trois dernières années, par chercheur et par an, 10 000 €. La dotation récurrente représente 4% des ressources totales, hors salaires et infrastructure du laboratoire.

Tableau II: Récapitulatif des contrats auxquels l'équipe MACAO participe.

Projet Durée Organisme	Durée	Organisme financeur	Partenaires	Thématiques
GALAXY	2010-2013 (4 ans)	ANR	Armines, LIP6, IRIT, Airbus, Akka, Softeam	Développement collaboratif de systèmes complexes selon une approche guidée par les modèles
SIRSEC	2009-2012 (4 ans)	DGE	Alstom, Thales, INRETS, CEALIST, IRIT, Serma Ingénierie, PrismTechn, Geensys	Système d'Information Reparti Sécuritaire
FullMDE	2009-2011 (3 ans)	ESA	Astrium Space Transport, Astrium Satellites, Esterel Technologies, IRIT, Praxis,VERIMAG	Full Model Driven Development for OnBoard Software
SoCKET	06/2008 – 06/2011 (3 ans)	DGE	Astrium Satellites, AIRBUS,CEALETI,CNES, IRIT, LabSTICC,Magillem Design Services, PSIS,ST Microelectronics, Thales R&T, TIMA	Soc toolKit for critical Embedded systems
DOMINO	03/2007 – 06/2009 (2,25 ans)	ANR	Airbus, CEA LIST, CNES, ENSIETA, INRIA, IRIT, Soft-Maint	Domaines et processus méthodologique
TOPCASED	08/2006	DGE, Région	Ada Core, Airbus, Anyware	Toolkit in open source for Critical
	12/2010 (4,2 ans)	MidiPyrénées	Technology, Astrium, Atos Origin, CNES, CS, ENSIETA, ESEO, FéRIA/LAAS, FéRIA/IRIT, FéRIA/CERT, INRIA, Micouin Consulting,	Application & Systems Development

			SiemensVDO, Sodifrance, Tectosages, Thalès	
TERESA	11/2009 – 11/2012 (3 ans)	7th FWP Europe	IRIT, Fraunhofer, Trialog, IkerlanK4, Escrypt, U.Siegen	Trusted computing Engineering for Resource constrained Embedded Systems Applications

I- INTRODUCTION

Le projet VeriFME financé par l'OSEO est un projet dont l'objectif concerne la validation de modèles. Il s'agit d'une validation formelle basée sur des vérificateurs de modèles (*model checkers*) pour la composante dynamique et sur l'outillage OCL pour la composante statique. De plus, il est envisagé d'introduire de la métrologie de modèles pour une validation quantitative des modèles.

Dans ce projet, deux des socles technologiques utilisés sont les plates-formes TOPCASED et NEPTUNE : deux logiciels qui possèdent tous les deux un interprète OCL. Celui de TOPCASED a été réalisé par IBM et celui de NEPTUNE par l'équipe MACAO de l'IRIT. Cette multiplication des interprètes OCL pose donc la question de leur conformité à la norme OCL telle qu'elle est définie par l'OMG, d'autant que la fondation Eclipse propose aussi son propre moteur OCL sous la dénomination Eclipse MDT/OCL. L'une des activités du projet VeriFME s'inscrit dans cette problématique de vérification exhaustive de la conformité d'un interprète OCL à sa norme.

Le travail à effectuer durant ce stage concerne la mise en œuvre des outils et méthodes nécessaires à la vérification de la conformité d'un interprète OCL à sa norme. Il se compose de quatre phases : la première consiste à étudier en détail la norme pour en extraire les points à vérifier. Cette étude porte aussi bien sur la syntaxe des règles que sur la vérification du typage que sur la conformité des bibliothèques. Il faut aussi tenir compte du fait que la conformité dépend de l'objectif de l'interprète selon qu'il est « FullOCL » ou qu'il est dédié uniquement à la vérification des méta-modèles. Cette différentiation est récapitulée dans la matrice de conformité présente dans la norme. A partir de ces éléments, la deuxième phase vise à proposer un modèle pertinent et les règles vérifiant chaque élément. Ces règles doivent être documentées afin d'indiquer précisément l'élément vérifié et sa référence dans la norme. Dans un troisième temps, un outillage et une démarche permettant de vérifier cette conformité doivent être définis afin de réaliser plus ou moins automatiquement la vérification grâce à un outillage pertinent. Cette phase nécessite l'étude des outils de tests existants comme Salome TM par exemple. Il est à remarquer que l'ergonomie et le retour des résultats aux utilisateurs est un point devant faire l'objet d'une attention particulière. Pour finir, la mise en œuvre de l'outillage et de la démarche pour vérifier doit être expérimentée sur les deux interprètes utilisés dans le cadre du projet VeriFME. Cette expérimentation permettra d'une part la mise à jour de la démarche et de l'outillage en prenant en compte les enseignements issus de

l'expérimentation et d'autre part permettra la définition d'une démarche générique pour la vérification d'outils à des normes.

Les normes évoluant, il est essentiel de proposer une solution facilement adaptable lors des évolutions normatives et il faudra en outre être en capacité de fournir rapidement un diagnostic indiquant les points non-conformes.

L'objet de mon stage étant de vérifier la norme OCL au niveau des outils NEPTUNE et TOPCASE, je présenterai tout d'abord les concepts importants se rapportant à OCL. Je ferai un état de l'art me permettant notamment de vérifier qu'il n'existe pas déjà sur le marché un outil permettant de le faire. Ensuite après avoir trouvé la démarche à suivre pour tester je ferai un état de l'art des outils de test. Je rentrerai ensuite dans le vif du sujet en étudiant quelles sont les règles à tester, quelle est la méthode à suivre, je choisirai les outils ensuite et je ferai l'automatisation et les tests.

Problématique

Il faut donc dans un premier temps vérifier la conformité à la norme des outils NEPTUNE et TOPCASED. Pour ce faire on pourra développer un outil indépendant qui permettra de tester les deux interpréteurs en s'appuyant sur la norme OCL.

Une première recherche à aboutit à la découverte d'un article très intéressant : une étude des résultats de l'évaluation de requêtes OCL à partir des différentes déclinaisons d'OCL, ceci nous intéresse pour vérifier la conformité d'OCL à la norme pour les outils NEPTUNE et TOPCASED [GKB08a]. L'auteur en est M. Gogolla.

Deux problèmes importants se sont alors présentés le fait que cet article traite essentiellement de la norme OCL 2.0 et que la version actuelle est OCL 2.3, et le nombre et la diversité des requêtes OCL à tester : il y a 950 règles dans l'article de Monsieur Gogolla et les règles présentées dans cet article ne sont pas exhaustives. Toutes les collections ne sont pas testées et le typage des expressions n'est que peu abordé. Il nous faut donc étudier ces règles manquantes. Pour le typage par exemple, si un entier est attendu et que l'on passe une chaîne de caractères, une erreur doit être générée.

Quantitativement et en se limitant aux seules règles proposées par M. Gogolla nous sommes confrontés a plus de mille règles à tester pour chaque outil soit deux milles règles en tout pour les deux outils. A raison du traitement de 6 règles à l'heure, cela nécessite plus de 300 heures de travail soit 10 semaines pour tester ces règles manuellement car nous travaillons 35 heures par semaine. Il faut donc trouver un moyen d'automatisation judicieux. De plus, ces tests devront être rejoués à chaque changement de version d'un outil. Dans le cas de TOPCASED, nous avons deux versions majeures par an.

Les tests présentés dans [GKB08a] ne prennent pas en compte la capacité des interprètes à traiter des modèles volumineux. Il faut donc aussi ajouter des tests de charge c'est-à-dire des tests permettant de valider le comportement des outils lors de la manipulation de gros volumes de données.

Pour ce faire on aura besoin d'utiliser un outil de test et d'organiser les tests en les partitionnant en campagnes de tests, chaque campagne traitant d'un aspect particulier à tester. Par exemple une campagne de test pour les collections et une autre pour les tests de charge. Ceci permettra ainsi de tester indépendamment chaque aspect à vérifier.

Quand on utilise des langages de programmation comme JAVA ou C on commence par faire la construction des modèles UML et on crée ensuite « manuellement » les programmes qui

obéissent à ces modèles. Les modèles représentent seulement une aide à la conception. Avec TOPCASED et NEPTUNE la conception des modèles est la base et l'origine de la conception et non plus seulement une aide à la conception. En effet les programmes Java sont alors générés automatiquement à partir des modèles.

TOPCASED et NEPTUNE utilisent des modèles UML comme des programmes à réaliser, générant automatiquement le code java. Dans ce contexte, il est nécessaire de posséder des outils de vérification, et validation de modèle. L'utilisation du langage OCL permet de s'assurer de la cohérence des aspects statiques du modèle. On devra donc créer un modèle dans un format compréhensible par chacun des deux outils, et un fichier contenant les règles OCL sur les modèles, à tester par les deux outils.

Il faudra alors analyser une à une les règles et pour chaque règle la soumettre aux deux interpréteurs OCL en tenant compte des outils considérés et en même temps analyser le résultat afin de vérifier la conformité à la norme OCL. Il nous faut donc d'abord présenter quelques aspects importants du langage OCL.

II-CONCEPTS

1-OCL

Object Constraint Langage est théoriquement une partie d'UML. Il permet de préciser des **contraintes** qui ne peuvent pas être exprimées à l'aide du langage de modélisation UML. Il est sans effet de bord c'est-à-dire qu'il ne modifie pas le modèle et permet de vérifier, entre autre, ce modèle par rapport aux contraintes exprimées dans son méta-modèle. Cette expression de contraintes s'applique indifféremment aux différentes couches de modélisation : au niveau du MOF, du méta-modèle (UML ou autre) ou du modèle (UML ou autre).

La vérification de la partie statique des modèles se fait grâce au langage OCL alors que la vérification de la partie dynamique est habituellement faite au travers de l'utilisation de « models checkers » qui sont des outils généralement basés sur des graphes, de la logique et des prouveurs basés sur les mathématiques.

OCL est un langage de **contraintes à objets** développé en 1997 par Jos Warmer (IBM). Il est basé sur la théorie des ensembles et la logique des prédicats. Il est développé sur les bases du

langage IBEL (Integrated Business Engineering langage). Il a été formellement intégré à UML 1.1 en 1999.
Voir le site http://www.lirmm.fr/~huchard/Enseignement/IDM/coursOCL20.pdf

Une contrainte OCL est liée à un contexte, qui est le type, l'opération ou l'attribut auquel la contrainte se rapporte.

context monContexte :
Expression de la contrainte

Tableau III : La classe Personne

Personne
- age : entier - majeur : booléen
+ getAge():entier {query} + setAge(in a : entier)

« **Expression de la contrainte** » peut prendre les valeurs suivantes :
– **inv** : pour exprimer un invariant de classe ;
– **pre** : pour exprimer une pré-condition sur une opération ;
– **post** : pour exprimer une post-condition sur une opération ;
– **body** : pour indiquer le corps d'une opération définie préalablement comme une « query » c'est-à-dire comme une opération sans effet de bord ;
– **init** : pour indiquer la valeur initiale d'un attribut ;
– **derive** : pour indiquer la valeur dérivée d'un attribut.

Par exemple, on peut définir les contraintes suivantes dans le **contexte** de la classe Personne du tableau III, puis dans le contexte de ses **méthodes** setAge, getAge et pour son **attribut** age.

```
context Personne inv :
(age <= 140) and (age >=0)
-- l'âge est compris entre 0 et 140 ans
```

Les attributs, comme par exemple « age » peuvent être de différents types comme :

- **Integer** : pour les entiers ;

- **Real** : pour les réels ;
- **String** : pour les chaînes de caractères ;
- **Boolean** : pour les booléens.

Quelques types spéciaux s'y ajoutent, en particulier :

- **OclModelElement** (énumération des éléments du modèle)
- **OclType** et maintenant **Classifier** (énumération des types présent dans le modèle)
- **OclAny** (tout type autre que Tuple et Collection hérite de ce type)
- **OclState** (état courant pour les diagrammes d'états)
- **OclVoid** sous-type de tous les types

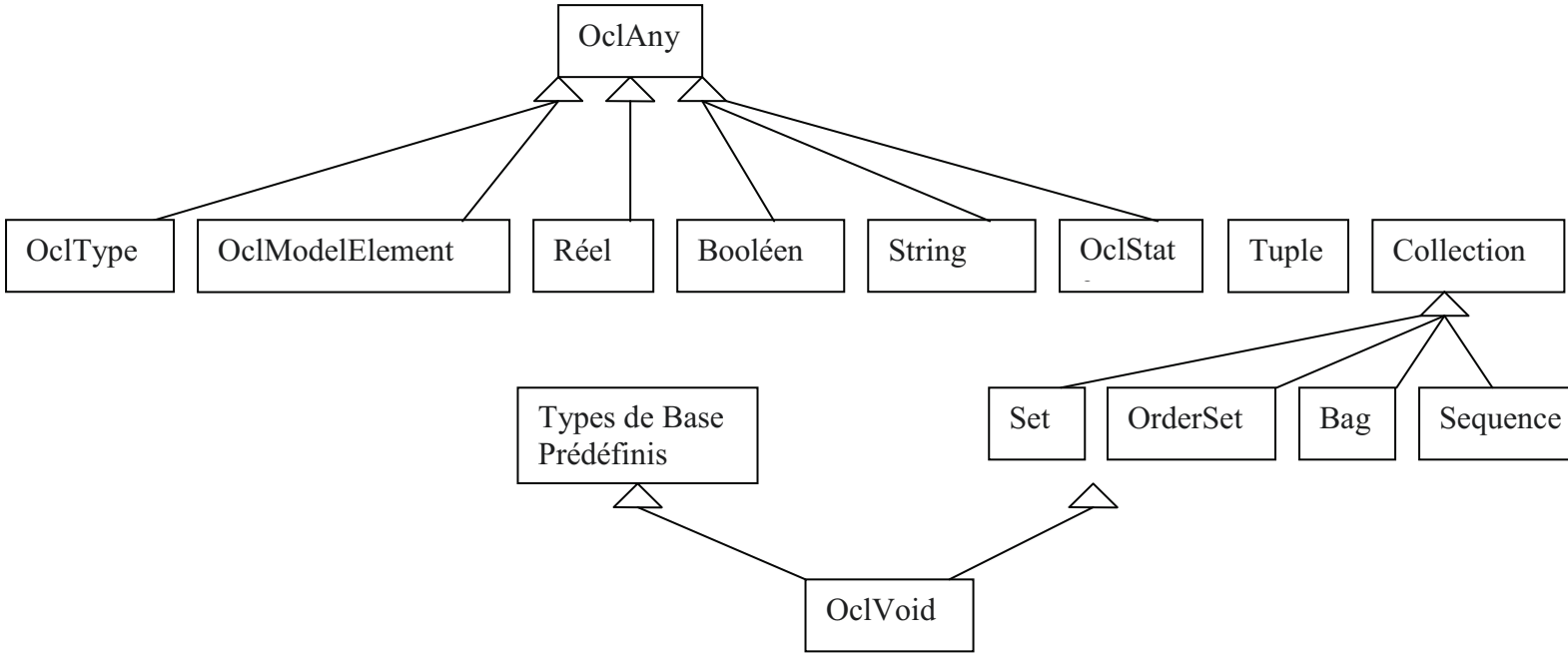


Fig. 3 : Types de base utilisés dans OCL

Une contrainte constitue une condition ou une restriction sémantique exprimée sous forme d'instruction dans un langage textuel qui peut être

- **naturel** : d'après la définition du lien ci-dessous « un langage ordinaire », une langue « normale » parlée par un être humain. En informatique, le langage naturel s'oppose au langage informatique.

- ou **formel**.

Voir le lien
http://www.bioinfo-biostats-etudiants.u-psud.fr/Ressources/Cours/Master%201/IG1/M1%20-%20cours%20MT1%20-%20Th%C3%A9orie%20des%20langages%20formels.pdf

Par exemple, la chaîne de caractères, placée entre accolades ({}), qui représente une contrainte dans un modèle UML, constitue le corps de la contrainte écrit dans un langage de contrainte qui peut être :

- naturel ;
- dédié, comme OCL ;
- ou encore directement issu d'un langage de programmation.

Si une contrainte possède un nom, on présente celui-ci sous forme d'une chaîne suivie d'un double point (:), le tout précédant le texte de la contrainte.

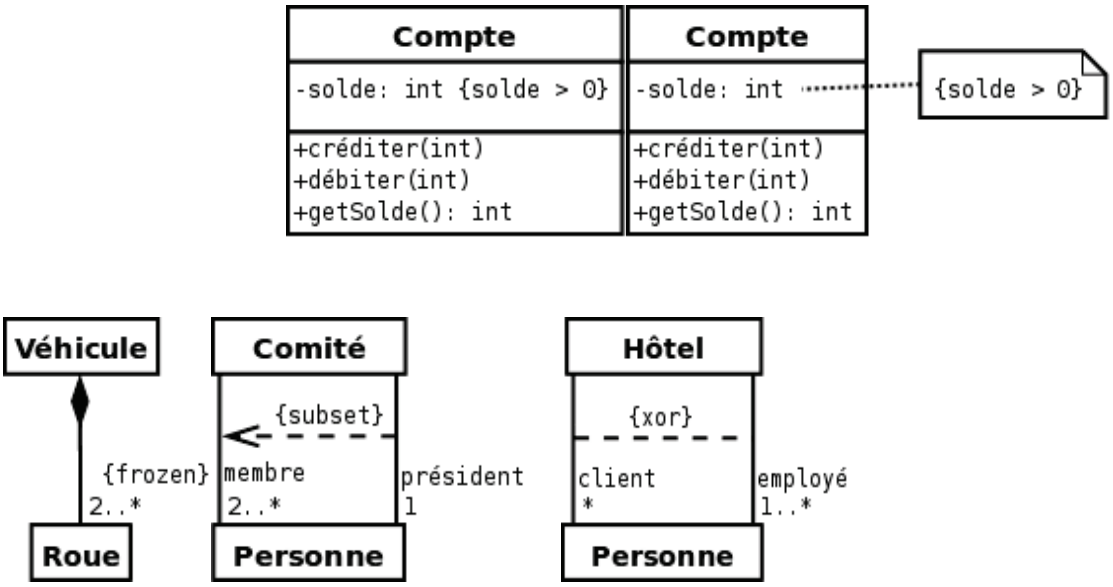


Fig. 4 : Exemple de présentation des contraintes prédéfinies(1)

Figure 4 : UML permet d'associer une contrainte à un élément de modèle de plusieurs façons. Sur les deux diagrammes du haut, la contrainte porte sur un attribut qui doit être positif. En bas à gauche, la contrainte {frozen} précise que le nombre de roues d'un véhicule ne peut pas varier. Au milieu, la contrainte {subset} précise que le président est également un membre du

comité. Enfin, en bas à droite, la contrainte {xor} (ou exclusif) précise que les employés de l'hôtel n'ont pas le droit de prendre une chambre dans ce même hôtel.

Fig. 5 : Exemple de présentation des contraintes prédéfinies(2)

Figure 5: Ce diagramme exprime que : une personne est née dans un pays, et que cette association ne peut être modifiée ; une personne a visité un certain nombre de pays, dans un ordre donné, et que le nombre de pays visités ne peut que croître ; une personne aimerait encore visiter toute une liste de pays, et que cette liste est ordonnée (probablement par ordre de préférence).

UML permet d'associer une contrainte à un, ou plusieurs, élément(s) de modèle de différentes façons (cf. figures 4 et 5) :

- en plaçant directement la contrainte à côté d'une propriété ou d'une opération dans un classeur ;
- en ajoutant une note associée à l'élément à contraindre ;
- en plaçant la contrainte à proximité de l'élément à contraindre, comme une extrémité d'association par exemple ;
- en plaçant la contrainte sur une flèche en pointillés joignant les deux éléments de modèle à contraindre ensemble, la direction de la flèche constituant une information pertinente au sein de la contrainte ;
- en plaçant la contrainte sur un trait en pointillés joignant les deux éléments de modèle à contraindre ensemble dans le cas où la contrainte est bijective ;

- en utilisant une note reliée, par des traits en pointillés, à chacun des éléments de modèle, subissant la contrainte commune, quand cette contrainte s'applique sur plus de deux éléments de modèle.

2 -SYNTAXE D'OCL

1-L'INVARIANT

- Un invariant est défini dans un **contexte**, le résultat attendu est un booléen : true ou false

  ex : `context Compte`

  Sa syntaxe peut être décrite comme suit : le mot « context » suivi d'un « élément »

  Nous noterons :
  context <élément>

  où <*élément*> est un nom de classe ou d'objet
  ex : **Person** respectivement **p : Person**.

- la syntaxe d'un **invariant** est la suivante :

  Inv <nom de l'invariant> : <expression logique>

  Où ***<nom de l'invariant>*** est juste une chaine de caractères qui spécifie le nom de l'invariant
  Ex : « enumGender »
  Ou « nameCapitalThenSmallLetters_VT »

  Dans ce cas précis l'invariant porte sur un nom qui doit commencer par une majuscule et être suivi de minuscules. VT représente le type de variable « Typed Variable »

  <expression logique> *est* une expression logique qui doit toujours être vrai si la contrainte est vérifiée. Elle peut porter sur des opérations portant sur des

collections, comme dans les règles OCL présentées dans le document de M. Gogolla. Le nombre de collections en OCL étant important il en résulte qu'un même invariant peut être défini de six façons différentes :

Pour commencer, il existe deux syntaxes pour allInstances (allinstances permet de sélectionner toutes les instances d'une métaclasse donnée) qui représente une collection : avec ou sans parenthèses :

Ex : Person.allInstances suivant OCL 1.x ou

Person.allInstances( ) suivant OCL 2.x

Il existe aussi trois syntaxes pour une même opération sur les collections selon que l'on utilise des variables ou non :

- *avec le type de la variable : le nom de la collection* (ex : **p : Person)**
- *sans le type de variable* : seulement le nom (ex : **p)**
- ou *sans le nom de la variable*

Par exemple ;

```
aPersonSet->select (p :Person|p.gender = 'female')
aPersonSet->select (p |p.gender = 'female')
aPersonSet->select (gender = 'female')
```

Nous constatons que les invariants sont en outre constitués d'attributs par exemple ici gender. Les attributs ont différentes formes.

2-LES ATTRIBUTS : 3 cas possibles :

- Soit c'est un attribut précédé de « self. » si dans le contexte l'élément est une **classe**

Ex : `self.gender` dans le contexte « Person »

- Soit si dans le contexte l'élément est un **nom d'objet**

Ex : **p** de type Person,

- Soit c'est le **nom d'un rôle**

`p.employee` où employee est le nom d'un **rôle porté par une association** liée à la classe Person.

Tableau IV : Types OCL de base et leurs valeurs

type	**valeur**
Boolean	true, false
Integer	1, -5, 2, 34, 26524, ...
Real	1.5, 3.14, ...
String	'To be or not to be...'

L'invariant est constitué d'attributs et d'**opérations** :

Tableau V : Examples d'opérations sur les types prédéfinis

type	**operations**
Integer	*, +, -, /, abs()
Real	*, +, -, /, floor()
Boolean	and, or, xor, not, implies, if-then-else
String	Concat(), size(), substring()

Ex :

```
substring(lower : Integer, upper : Integer) : String
```

Le sous-champ de la chaîne commençant au caractère *lower*, jusqu'à et incluant le caractère de rang *upper*. Le nombre de caractères va de 1 à *self.size()*.

```
pre: 1 <= lower pre: lower <= upper pre: upper <= self.size()
```

Maintenant que nous avons évoqué OCL qui est le cœur de mon mémoire, nous allons faire l'état de l'art des logiciels supportant le langage OCL puisque c'est l'objet de notre étude.

III-ETAT DE L'ART

1-ETUDE DES OUTILS DE TRAITEMENT D'OCL

Après l'étude des outils suivants USE, OCL Dresden Toolkit, Octopus, MDT OCL, Together, XMF, OCLE, et KMF (voir article [CPP08]), l'outil idéal aurait les caractéristiques suivantes :

- le langage de modélisation aurait un formalisme textuel qui complèterait le modèle ;
- il devrait permettre la navigation dans le modèle et la navigation dans les spécifications d'OCL ;
- Il devrait être conforme aux règles de construction de son langage de modélisation WFR (règles spécifiées au niveau du métamodèle). Pour cela il doit être possible de compiler OCL et implicitement de l'évaluer).

Ce que ne fait pas OCL en règle générale c'est-à-dire en considérant les outils OCL existants (plusieurs outils sont des extensions d'OCL utilisés pour divers domaines d'application) :

- ✓ les évaluations des expressions contenant « Undefined Values ». Par exemple quelques expressions, quant elles sont évaluées prennent la valeur « Undefined ». Par exemple, le transtypage avec oclAsType() vers un type qui prend le premier élément d'une collection vide->first() élément va donner comme résultat « Undefined ». En général une expression dont un des éléments est « Undefined » va avoir elle-même comme résultat « Undefined ».

- ✓ l'évaluation des fonctions non déterministes suivantes : Any, asSequence, asOrderedSet, (page 118 de la norme OCL) IterateExpEval**.**
  Du point de vue du concepteur, le standard OCL donne une spécification incomplète pour l'opération
  any(iterator|<boolean expression>) définie pour les collections.
  De ce fait les résultats attendus quand on évalue une expression OCL dépendent de l'outil utilisé.

Avec USE

`Set{9,1,7,oclUndefined(Integer)}→ any(true) = oclUndefined(Integer)`

Avec OCLE

`Set{9,1,7,oclUndefined(Integer)}→ any(true) = 1`

- ✓ l'accès aux propriétés qui ont le même nom dans les ascendants.
  L'accès aux propriétés d'héritage vers le bas est possible, mais l'accès vers le haut est impossible. (Voir page 182 Full Descriptor of a Class dans [OMG09]). La norme dit que les enfants héritent des propriétés de leurs parents mais c'est tout.

- ✓ au niveau du MOF les associations qualifiées et les associations de classes sont manquantes : comparé au langage UML les associations sont binaires et non n-aires (n>2 page 18 **Missing AssociationEnd names** de [OMG09], page 19 **Navigation over Associations with Multiplicity Zero or One** de [OMG09]**)**

- ✓ il y a des différences entre les outils dans la manière d'implémenter la norme OCL.
  - ✓ Toutes les différences doivent être clairement et explicitement décrites
  - ✓ les standards doivent être testés validés et améliorés avec des outils appropriés
  - ✓ l'outil devrait permettre de travailler avec des modèles de n'importe quelle taille.
  - ✓ on devrait pouvoir travailler avec des expressions OCL à trois niveaux d'abstraction :
    - Méta-méta-modèle
    - Méta-modèle
    - Et modèle

A l'heure actuelle les outils qui me semblent les plus intéressants sont « Dresden OCL toolkit » et « USE » de part leurs fonctionnalités OCL et le fait qu'ils sont constamment améliorés. La différence entre les deux : l'un est spécialement conçu pour être intégré à d'autres outils de modélisation et à des IDE : il s'agit de « Dresden OCL toolkit » alors que l'autre est un outil indépendant : « USE » :

« Dresden OCL toolkit » comprend un compilateur OCLCUD6 qui permet de vérifier la syntaxe et la sémantique des expressions OCL. Il traduit aussi les contraintes en JAVA et en SQL. Il peut être utilisé comme applet de démonstration du compilateur OCL ou comme partie d'Argo/UML.

OCLCUD6 est composé de 4 modules :

- Un parseur
- Un analyseur de sémantique
- Un outil de normalisation
- Un générateur de code.

Il faut charger le modèle UML pour analyser les contraintes OCL, ce modèle peut être obtenu avec un fichier XMI généré par exemple par ArgoUML.

.

OCLCUD6 permet les vérifications suivantes :

- L'expression a un contexte
- La vérification de la cohérence des contraintes par rapport au modèle
- La simplification (pour réduire la complexité du code) puis la génération de code.

OCLCUD6 possède les avantages suivants :

- La possibilité de travailler avec des fichiers XMI ;
- Le haut degré de communication avec les outils de conception ;
- L'analyseur est adaptable à une importante variété d'objectifs, le plus important étant l'intégration du compilateur OCL dans l'IDE ArgoUML qui est en open source (cet outil est toujours en cours de développement).

Mais OCLCUD6 a aussi des limitations :

- Certaines contraintes OCL ne peuvent pas être utilisées comme celles utilisant l'operation notEmpty() ;
- L'analyse sémantique est limitée à la cohérence d'expressions et à la vérification des types ;
- La génération de code accepte seulement un certain langage normalisé qui simplifie et complète les expressions OCL (voir l'article [TRF03]).

« USE » est l'outil le plus rapide en ce qui concerne l'évaluation des expressions OCL sur les scénarios moyens et grands sachant que le temps nécessaire pour traiter les opérations dépend de la taille des collections (voir l'article [CEG08]). « USE » permet de faire l'analyse syntaxique, la vérification de type, la cohérence des contraintes, l'évaluation des prés et post conditions.

Mais « USE » a un inconvénient majeur : l'outil utilise un langage spécifique pour représenter les modèles UML qui n'est pas compatible avec XMI. Toutefois il est prévu pour les versions futures le support du standard XMI.

Pour avoir une vue d'ensemble des différents outils utilisant OCL, le tableau suivant résume les fonctionnalités des outils « ModelRun », « OCL Compiler de Cybernetic », « OCL Compiler de Dresden », « OCL Checker » et « USE » et compare leurs fonctionnalités.
On peut constater que :

- « ModelRun », « OCL Compiler de Dresden » et « USE » possèdent l'option de vérification de type ;
- Seuls « OCL Checker » et « OCL Compiler de Cybernetic » possèdent un outil indépendant de l'édition des modèles.
- Seuls « ModelRun » et « OCL Compiler de Dresden » utilisent respectivement XML et XMI pour les le chargement de modèle UML alors que « USE » possède sa propre syntaxe de connexion avec le modèle UML.
- « ModelRun » et « OCL Compiler de Dresden » possèdent un guide d'assistance aux utilisateurs alors que « OCL Compiler de Cybernetic », « OCL Checker » et « USE » non.
- Seul « OCL Compiler de Dresden » génère du code Java automatiquement dans cette étude comparative.
- Seuls les outils « ModelRun » et « USE » font une évaluation dynamique des invariants.
- Seul « USE » fait une évaluation dynamique des pré et post-conditions.

Le tableau VI date de 2003, depuis OCL Compiler de Dresden fonctionne avec la version 2.2 d'OCL, les autres outils fonctionnent toujours avec les anciennes versions d'OCL.

Tableau VI : Etude comparative des différents outils à base d'OCL extrait de [TRF03] (2003)

Table 2. Summary of the comparison

	ModelRun	OCL Compiler (Cybernetic)	OCL Compiler (Dresden)	OCL Checker	USE
Syntactic analysis	Yes	Yes	Yes	Yes	Yes
Type checking	Yes	No[1]	Yes	No	Yes
Model-independent tool	No	Both uses[2]	No	Yes	No
Connection with UML model	XML file	"front-end" tool	XMI file	–	Own file
Guided support	Contextual syntax assistant	No	Syntax assistant	No	No
Code generation	No	No	Java, SQL	No	No
Introducing OCL expressions	Import model and internal editing	Import model and internal editing	Import model and internal editing	Import from file	Import from file and internal editing
Dynamic validation of the invariant	Yes	No	No	No	Yes
Consistency checking	Constraint compatibility	Constraint compatibility[3]	Constraint compatibility	No	Constraint compatibility
Dynamic pre-/ postconditions validation	No	No	No	No	Yes
Version OCL	OCL 1.3	OCL 1.3 and OCL 1.3 extended	OCL 1.3 extended	OCL 1.4 pre-release	OCL 1.3

Vu la relative spécialisation de chaque interpréteur OCL, il est difficile de dire lequel est le meilleur. Pour quelqu'un qui débute et qui veut un outil simple à utiliser, rapide et qui vérifie simplement la syntaxe des expressions, il vaudra mieux utiliser « Model Run » ou « OCL Checker » de J.Warmer.

Si c'est pour vérifier les contraintes d'un diagramme UML, un outil incluant un support pour les diagrammes UML, pourra être choisi. Dans ce cas là il faudra prendre en compte les aspects suivants :

Est-ce qu'on veut un simple outil d'analyse, juste l'analyse syntaxique et la compatibilité des contraintes ou un outil plus complet de validation du modèle.
Dans le premier cas on pourra utiliser « OCL compiler » de Cybernetic. Sinon « ModelRun » de Bold-Soft qui inclut une fonctionnalité d'animation du modèle UML, ou bien « USE » de M. Richters et M. Gogolla.
Mais le seul outil, dans cette étude comparative, capable de générer automatiquement le code JAVA et SQL à partir du modèle est « OCL Compiler » de l'Université de Dresde de plus c'est le seul qui a évolué car il utilise la version 2.2 d'OCL, donc c'est le plus à jour.
Les deux aspects à privilégier pour choisir un outil est de prendre celui qui peut importer des modèles UML en XMI ou XML, et celui qui offre la meilleure documentation pour générer les contraintes. (Voir l'article [TRF03])
D'autres outils existent comme ATL OCL, OCTOPUS, RoclET, Kermeta OCL, KMF, OSLO, VMTS OCL. (Voir l'article: [GKB08]) mais nous ne les aborderons pas ici car ils nous ont semblés moins intéressant pour notre problématique.

CONCLUSION

Dans cette étude comparative, les outils « Dresden OCL toolkit » et « USE » sont les plus intéressants et permettent notamment la vérification du typage, mais seul « Dresden OCL toolkit » travaille avec XMI ce qui pour l'objet de mon étude concernant la vérification du respect de la norme OCL par TOPCASED et NEPTUNE est le plus adapté. De plus c'est le seul à utiliser la version 2.2 d'OCL Ils fonctionnent tous les trois avec des fichiers XMI. De plus « Dresden OCL toolkit » génère du code JAVA et SQL mais il ne valide pas les invariants et les prés et post-condition ce que fait « USE ».
Comme « Dresden OCL toolkit » m'a paru l'outil le plus complet, j'ai voulu l'étudier plus précisément.
Au départ je voulais faire les tests des interprètes OCL sur les trois logiciels TOPCASED, NEPTUNE et DRESDEN OCL TOOLKIT, (pour faire une étude plus complète). J'ai donc téléchargé DRESDEN OCL via ECLIPSE HELIOS (documentation : [WTF11] DRESDEN OCL Manual for Installation, Use and Development) et étudié comment il fonctionnait grâce à l'exemple qui y figure. Toutefois par manque de temps j'ai dû me résoudre à recentrer mon étude sur NEPTUNE et TOPCASED. Néanmoins, j'ai tout de même évalué DRESDEN OCL

pour me familiariser avec les outils OCL et ainsi mieux appréhender les deux autres outils. Pour réaliser ces tests j'ai :

- Ouvert la fenêtre Dresden OCL (Windows-> Open Perspective -> Other puis sélection de Dresden OCL)
- J'ai chargé le modèle (File->New->Other puis j'ai choisi l'exemple du modèle Dresden ocl : option DRESDEN OCL examples -> SIMPLE EXAMPLE
- J'ai chargé ensuite le modèle UML (sélectionner model/simple.uml puis clic droit et Dresden OCL-> Load Model), ainsi dans le Model browser j'ai visualisé une instance du modèle UML

Puis j'ai affiché le détail du programme java ModelProviderClass.java qui spécifie les classes du modèle

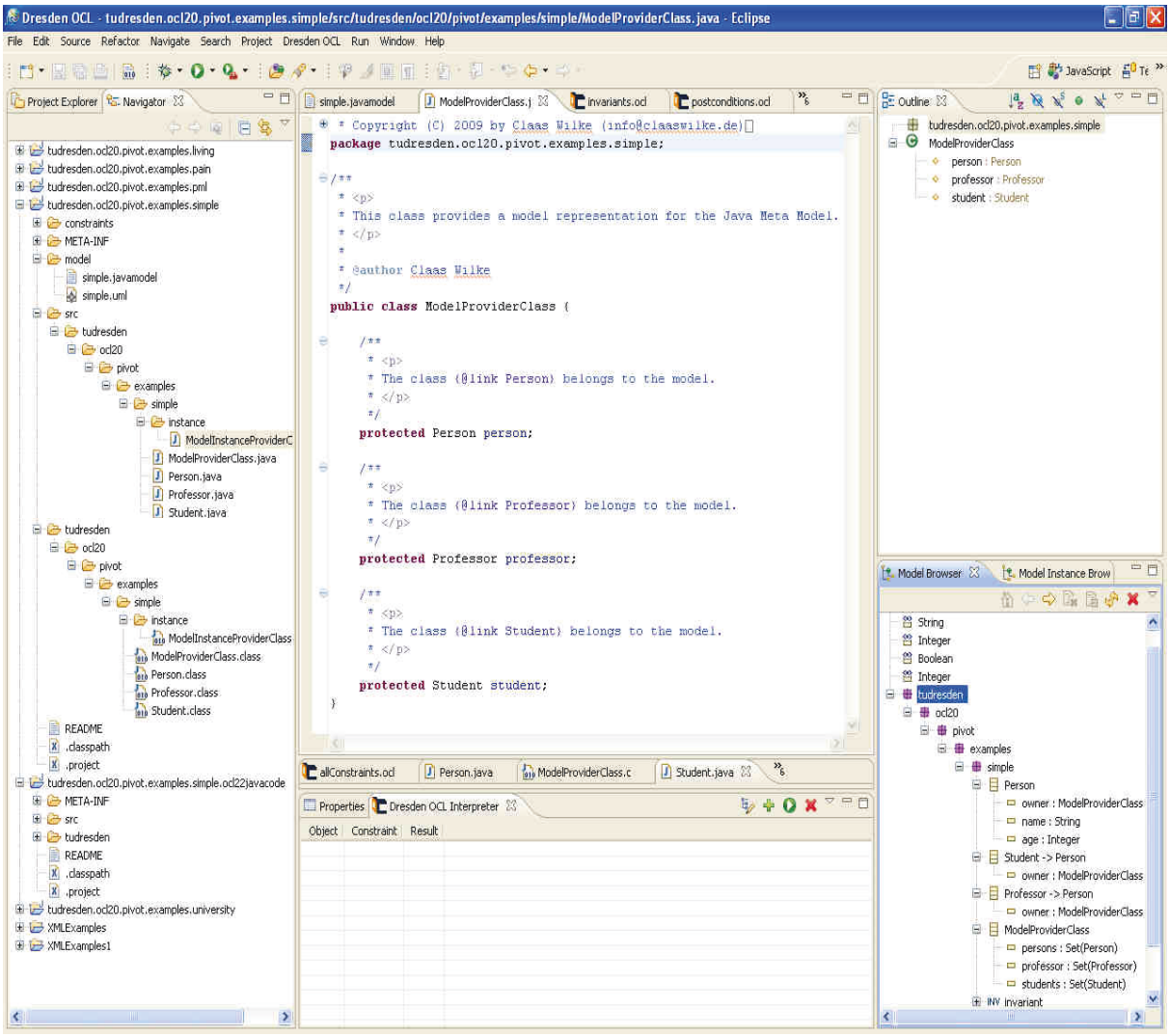


Fig. 6 : Aperçu de l'outil Dresden OCL

La fenêtre « Navigator » à gauche permet la navigation dans les différents projets. J'ai choisi par exemple d'explorer le projet « tudresden.ocl20.pivot.examples.simple ». Il contient dans son répertoire « constraints » les différentes contraintes OCL (fichiers d'extension .ocl), dans le répertoire « model » les modèles Java et Uml, dans le répertoire « src » qui signifie source, les sources des programmes Java générés automatiquement (les fichiers d'extension .java) à partir des règles OCL dont on peut voir le contenu dans la fenêtre du milieu : voir le contenu de « ModelProviderClass.java ». Au-dessous la liste des programmes exécutables (d'extension .class). La figure 7 est un complément de la figure 6.

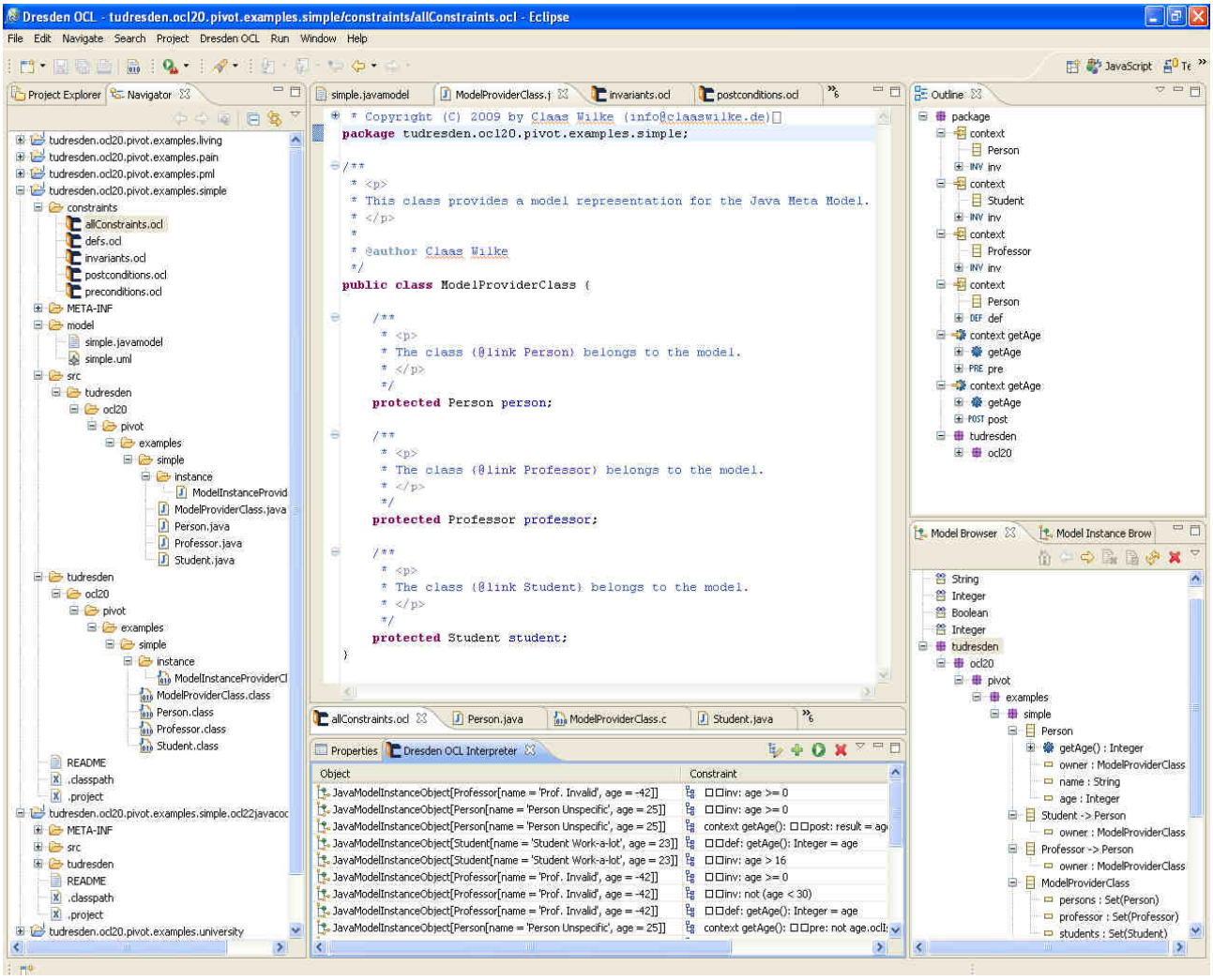


Fig. 7 : Aperçu de l'interpréteur OCL de l'outil Dresden OCL(1)

Dans la figure 7 le cadre central du bas nommé « Dresden OCL Interpreter » contient le détail de l'exécution des contraintes OCL avec dans la colonne de gauche le résumer d'une instance du modèle et dans la colonne de droite l'invariant à appliquer à cette instance.

En bas à droite de la figure 7 apparait le « Model Browser » qui détaille les différentes classes UML (ex : « Person ») avec les attributs (ex : name) et les opérations (exemple : getAge())

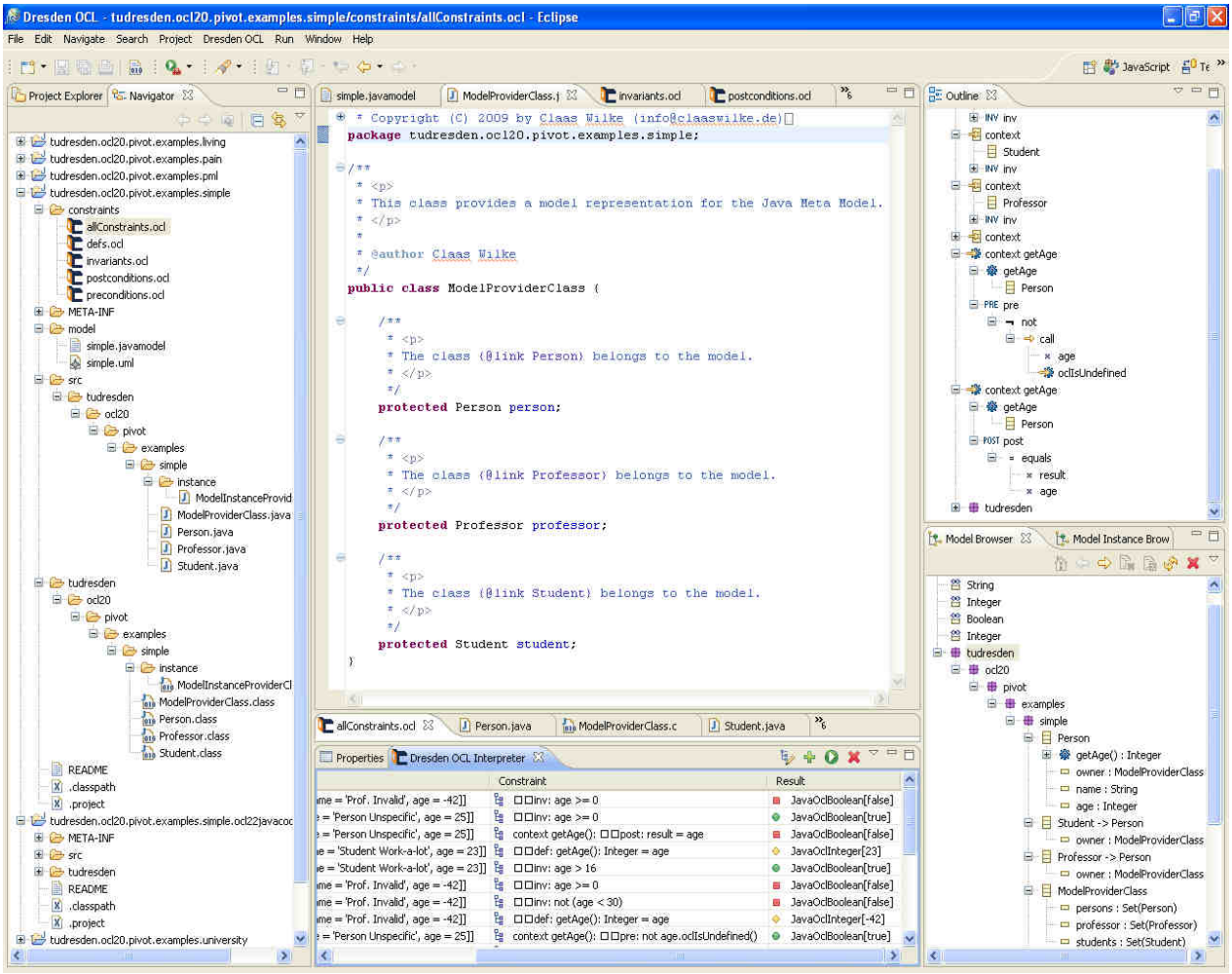


Fig. 8 : Aperçu de l'interpréteur OCL de l'outil Dresden OCL(2)

La figure 8 est plus complète au niveau du cadre « dresden OCL Interpreter », on y voit clairement le résultat de l'exécution des requêtes OCL (true ou false par exemple, ou 23 pour l'âge de la personne)

Maintenant que nous avons étudié les logiciels existants sur le marché et comme nous l'avons vu dans la problématique, nous allons avoir à tester différents aspects de ces logiciels et plus précisément leur interprète OCL et donc réaliser des tests sur de gros volumes de tests. Il nous faut donc un outil de test adapté. Nous allons faire l'état de l'art des outils de test.

2- ETUDE DES OUTILS DE TEST

A- Les types de test

La création d'un logiciel se fait en 3 phases : la conception, la réalisation et les tests.
A l'heure actuelle le découpage de cette activité est fait ainsi :

- la vérification des exigences 40%
- une première phase de conception 30%
- la conception détaillée et les tests 30%

Car un logiciel mal développé nécessite un coût de maintenance élevé, dans les tests il est important de prévenir un maximum les retours négatifs de l'utilisateur.

L'outil de test doit donc prendre en compte les tests suivants :

- les tests de « **Boîte Noire** » ou « **Black Box** » qui testent le fonctionnement externe du logiciel c'est-à-dire la vérification des sorties par rapport aux entrées
- les tests de « **Boîte Blanche** » ou « **White Box** » qui testent le comportement interne du logiciel, ils vérifient que toutes les lignes du programme ont été exécutées
- les tests de **conformité** du logiciel par rapport à ses exigences fonctionnelles ou techniques ce qui revient à tester les cas classiques d'utilisation.
- les tests de **non-conformité** qui contrôlent tous les cas ne faisant pas partie des exigences afin de vérifier qu'ils ne perturbent pas le fonctionnement du logiciel.

Généralement les tests sont regroupés en niveaux comme suit :
-***Tests unitaires*** qui permettent de détecter les erreurs techniques d'un programme.
-***Tests d'intégration*** : test de plusieurs fonctionnalités qui s'enchaînent.
-***Tests fonctionnels*** : qui assurent que l'ensemble des exigences et des fonctionnalités requises dans les spécifications sont présentes, fiables et performantes. Il s'agit de tests poussés, réalisés directement sur l'interface de l'utilisateur final prenant en compte les tests de conformité et de non-conformité aux exigences.

-***Tests de pré-production ou de validation*** qui permettent de vérifier que l'application informatique fonctionne correctement sur son environnement de production c'est-à-dire en parallèle d'autres applications avec une volumétrie proche de celle de la production.
-***Tests de recettes*** réalisés sur un échantillon représentatif d'utilisateurs finaux qui vérifient que le logiciel correspond à leurs attentes.
-***Tests de non régression*** : ils permettent de s'assurer que la correction d'erreurs n'en a pas généré de nouvelles.

Nous verrons que dans notre étude, les tests que nous ferons peuvent s'apparenter aux tests de conformités aux exigences qui sont les tests d'utilisation classique (c'est-à-dire tester si les invariants sont vérifiés) et les tests de non-conformité qui sont les cas ne faisant pas partie des exigences comme les tests de charge par exemple. Pour cela nous choisirons parmi les différents outils de test celui qui semble le plus adapté à notre étude.

B – Etat de l'art des outils de test

Il existe sur le marché de nombreux outils de test, nous allons en étudier sommairement quelques-uns et choisir celui qui est le plus adapté à notre étude.

HP Quality Center

A l'origine créé par la société "Mercury Interactive", cet outil de test appartient désormais à « HP », il est considéré comme le leader sur le marché des outils de test.
Cet outil assure la qualité logicielle, la gestion des exigences, les tests fonctionnels et permet notamment de tester des modules SAP. Dans ces fonctionnalités nous trouvons notamment la possibilité de :

- Générer des tests manuels et automatisés
- Fournir des processus cohérents répétables pour obéir aux exigences
- Analyser les résultats
- Gérer les erreurs
- Communiquer entre les équipes
- Appliquer les méthodes Agiles

Coût : il est payant
Dimensionnement du projet testé : grand

IBM Rational Testing

Est le dauphin de l'outil « HP », cette solution est récente sur le marché des outils de test. Elle est basée sur une architecture particulière, composée d'une application centrale à laquelle sont connectées d'autres applications ajoutant des fonctionnalités supplémentaires.
Coût : il est payant
Dimensionnement du projet testé : grand

QaTraq Professionnal

Conçu principalement pour les responsables d'entreprise, cet outil met en avant l'édition de graphiques et de suivi de test.
Coût : il est payant
Dimensionnement du projet testé : moyen

Bugzilla Testopia

Testopia est une extension de l'application Bugzilla qui recueille et suit les bugs relevés dans une application. Enrichi de ce plug-in il permet principalement d'effectuer les tests unitaires et d'intégration.
Coût : il est gratuit
Dimensionnement du projet testé : moyen

Test run

Edité par "Majordojo" Test Run propose une interface destine aux chefs de projets. Particulièrement axé sur les niveaux de test, cet outil est le seul à en faire la distinction explicite. Il propose en outre des rapports complets.
Coût : il est payant
Dimensionnement du projet testé : petit

Test link

Proposé en "Open Source" par l'éditeur TEAMST, il met à disposition une interface plutôt soignée.
Coût : il est gratuit
Dimensionnement du projet testé : petit

Salome TMF

Développé par la branche recherche et développement de la société ORANGE/France Telecom. Il est écrit en JAVA et PHP, utilise une base de données MySQL et un serveur APACHE. De plus il est distribué en open source.
Il fonctionne avec des plug-ins de :

- Gestion des exigences (Requirements) c'est-à-dire les tests de conformités aux exigences. Une fois les tests effectués un graphique affiche le pourcentage d'exigences couvertes par les tests.
- Gestion des anomalies (MANTIS) : Une anomalie est un rapport d'erreurs créé suite à l'échec d'un test qui permet de tracer celui-ci et de demander la correction du problème.. La première étape consiste à ajouter le projet courant dans la base de données des bugs via Mantis. Après l'utilisateur peut saisir et consulter une anomalie relative à un test. Dans la saisie il peut indiquer par exemple la gravité du bug, le destinataire (celui qui va traiter le bug), la priorité avec laquelle ce bug doit être corrigé puis un résumer et une description du bug. Il est possible aussi de consulter les anomalies relatives à un environnement. Pour cela il faut cliquer sur un environnement dans la fenêtre des environnements puis cliquer sur « gestion des anomalies » et choisir l'option « Mantis » et « Affiche les bugs ». La liste des bugs apparait alors.
- Gestion de la documentation (GENDOC) : L'utilisateur peut générer des documents de test ou sont tracés les résultats des tests contenant toutes les informations sur les campagnes, avec les fichiers attachés aux environnements (les fichiers d'extension .bat . exe .class .java .jar), les paramètres de test et leur valeur, les différents environnements de test avec leurs paramètres, les scripts de test, les dossiers de tests avec par campagne et par test, le nom du test, sa description le résultat attendu et le résultat obtenu.
- Gestion des tests automatiques : il permet de rejouer les tests de non-régression.
- Il permet aussi de faire des traitements différés, lancer des batch de nuit pour traiter de grosses quantités d'information.

Coût : il est gratuit
Dimensionnement du projet testé : petit

J'ai choisi cet outil car il m'a paru complet (grâce à de nombreux plug-in), la documentation utilisateur est assez fournie, il m'a paru avoir une bonne interface utilisateur et pour le type de test que j'ai à faire il me semblait approprié.
En effet, chaque test aura les caractéristiques suivantes :

- Données en entrée
- Actions utilisateur

- Données attendues
- Données réellement retournées
- Statut du test

Pour cela l'outil installé devra être stable, ergonomique et performant, cependant sa complexité devra rester en rapport avec la complexité du logiciel.
Les cas les tests se borneront à considérer la valeur retournée par l'invariant « true » « false » ou « Undefined » et à notifier les arrêts du programme qui seront dans notre cas dus à des problèmes de syntaxe d'OCL, ces arrêts pourront être tracés grâce au plugin Mantis de Salome TMF

J'ai donc décidé de l'installer et de le tester. Pour ce faire, j'ai installé cet outil et constaté qu'il fallait utiliser une ancienne version de Java (la version Jdk1-6.0-17) pour qu'il fonctionne ce qui est un peu gênant. De plus la dernière version de Salome TMF qui peut s'installer automatiquement est la version 3 qui date du 20/12/2007, les autres versions plus récentes requièrent une installation manuelle avec tous les risques d'erreurs dans la configuration qui peuvent se produire. L'utilisation de plugins dont la version doit être antérieure à celle de Salome TMF version 3 et l'installation des plugins ne va pas de soi. Notamment l'installation du plugin Mantis tel quel ne marche pas. Le fichier mantis.jar n'était pas le bon, il fallait télécharger ce fichier à l'adresse suivante :
http://forge.ow2.org/project/download.php?group_id=194&file_id=9757.
De plus Salome TMF est « une usine à gaz », il met environ dix minutes à se charger lorsqu'on ouvre l'application.
J'ai été confrontée à des problèmes techniques que j'ai résolus en lisant les forums correspondants sur Internet. De plus il existe un plugin « Helpgui » qui est chargé avec Solome TMF et qui explique toutes les manipulations à faire pour installer les plugins et utiliser Salome TMF. Toutefois celle-ci est disponible uniquement après que le travail d'installation soit fait. Il vaudrait mieux qu'il soit disponible sur Internet, cela faciliterait le travail.
Important : Le fichier mantis.jar n'est pas bon, le fichier correct est à télécharger à l'adresse suivante : http://forge.ow2.org/project/download.php?group_id=194&file_id=9757

CONCLUSION SUR L'ETAT DE L'ART:

Le choix de l'outil de tests doit se faire en fonction du type de tests que l'on veut réaliser s'il s'agit de faire uniquement des tests fonctionnels c'est-à-dire des tests de conformité et de non-conformité aux exigences seuls les outils « IBM Rational Testing » et « Bugzilla Testopia » le permettent. En ce qui concerne ma problématique, « Salome TMF » m'a semblé le plus complet car possédant un nombre important de plugin et une documentation utilisateur assez complète, par la suite j'ai vite été confrontée à de nouvelles difficultés car en plus des différents problèmes énoncés précédemment, il nécessite notamment l'utilisation du langage Beanshell [MVM10] pour le lancement des scripts, et il existe très peu de documentation se rapportant à Beanshell. Nous avons dû en commander un [MVM10] (Voir « Etude de Salome TMF et comparaison avec d'autres outils de tests » de Gildas LAVERGNE [Lav10]).

IV –SOLUTIONS PROPOSEES

L'objet de mon stage à l'IRIT est de vérifier la conformité des interprètes OCL des outils TOPCASE et NEPTUNE à la norme car à ce jour aucun outil n'existe qui met en œuvre l'intégralité de la norme. Dans ce contexte, il m'est demandé de sélectionner voire de proposer un outil pertinent pour vérifier la conformité à la norme.

Pour ce faire, l'étude des documents: [GKB08a] et [GKB08b] met en exergue les points à vérifier du langage OCL c'est-à-dire les tests à effectuer au niveau de NEPTUNE et de TOPCASED pour vérifier la conformité par rapport à la norme.

Pour cela, dans [GKB08a] et [GKB08b], M. Gogolla a regroupé les tests à faire selon sept catégories. Ce découpage a pour objectif de tester les différentes parties de la norme OCL. Ces parties numérotées de B1 à B7 regroupent les tests sur les points suivants et sont détaillées dans [GKB08b]. J'ai fait la traduction de la description des différents items du document [GKB08b].

- B1 : les types de données, les invariants, les propriétés, les associations
- B2 : les énumérations, les pré- et post conditions, les requêtes
- B3 : les associations ternaires et les associations de classes
- B4 et B5 : les opérations et les propriétés des collections
- B6 : conformité sur les caractéristiques non déterministes « any, flatten »
- B7 : efficacité : évaluation du type de données, les opérations sur les collections

Les parties B1 à B5 testent la conformité d'OCL à la norme et plus particulièrement les caractéristiques syntaxiques et sémantiques.

Les contraintes OCL et les requêtes sont exprimées dans un contexte comme par exemple une classe ou une opération. Pour cela un moteur OCL doit être un support au langage de modélisation. Les caractéristiques les plus courantes sont les diagrammes de classes et d'objets pour les évaluations de dépendance d'état.

L'étude [GKB08b], vérifie que le MOF central et respectivement les caractéristiques des classes, attributs, énumérations et associations des classes UML sont cohérents.

Dans notre cas, le centre de l'étude est par exemple les propriétés sur les objets (les attributs et les rôles), les opérations sur les collections et les navigations avec notamment l'utilisation de allInstances(). Un moteur OCL doit être capable d'évaluer des expressions dépendantes

d'un contexte : (ex: `Person.allInstances()->select(age>18)`) et les expressions indépendantes (Ex: `set{1..9}->collect(i|i*i)`).

Comme il est indiqué dans le standard OCL, l'évaluation des requêtes en retournant une valeur et un type pour le résultat est une tâche importante du moteur d'OCL. Les expressions dépendantes sont liées aux objets, à leurs attributs et à leur rôle. Typiquement ce genre d'expression sont utilisées dans les prés et post-conditions d'OCL spécifiant ainsi les effets induits par les opérations et les invariants d'OCL.
L'étude comparative [GKB08b] couvre tous les éléments OCL mentionnés.
La version 1.3 et 2.0 montre des différences mineures pour certaines constructions syntaxiques. Par exemple suivant OCL 1.3 toutes les instances d'une classe sont retrouvées avec `allInstances`, alors que dans OCL 2.0 `allInstances()` est utilisé. Cette étude traite des contraintes sur des variations particulières de syntaxe dans le but de tester les versions 1.3 et 2.0 d'OCL.
En parallèle d'une vérification de la complétude des caractéristiques d'OCL, une évaluation correcte et consistante des contraintes OCL et des requêtes est requise. La base d'une évaluation exacte d'une expression complexe est une implémentation correcte de chaque opération OCL individuellement. De tels tests sont mis en pratique, en appliquant des opérations OCL sur les collections, sur les types de données des opérations et sur les littéraux d'énumération dans les expressions complexes.

- **B1 : Etude comparative du cœur**

Le cœur de l'étude vérifie OCL et les caractéristiques courantes du langage de modélisation.

En ce qui concerne le langage de modélisation, le modèle appliqué inclut une classe avec des attributs, une opération qui en découle et une association binaire réflexive comme illustré sur la figure suivante :

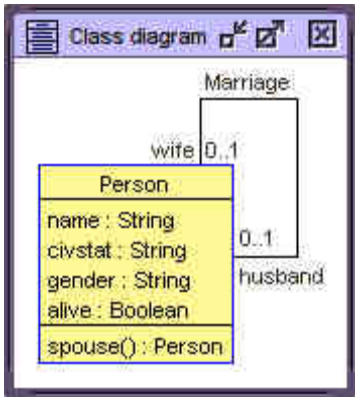


Fig. 9. Diagramme de classe du modèle e-core

L'étude B1 évite le traitement des caractéristiques spéciales et avancées comme les énumérations, les collections vides, et les « `undefined values` » et fournit plusieurs syntaxes différentes pour une même expression.

Les opérations couramment utilisées dans OCL et les constructions sont ajoutées au modèle à travers les invariants, les prédicats booléens de base, les opérations collect[1] et flatten[2], l'expression let[3], les collections imbriquées c'est-à-dire les collections qui contiennent des collections, et la navigation avec le raccourci collect.

Utiliser le raccourci `collect` revient à appliquer une propriété à une collection d'objets.

Par exemple la propriété .name s'applique à la collection `Person.allInstances()et` s'écrit `Person.allInstances().name` ce qui revient à écrire `Person.allInstances->collect(name).`

---

[1] Collect permet de construire une nouvelle collection en utilisant la collection *self*. La nouvelle collection construite possède le même nombre d'éléments que la collection *self*, mais le type de ces éléments est généralement différent. La syntaxe de l'opérateur *collect* est la suivante :
`Collect ( [ <élément> [ : <Type> ] | <expression> )`
voir page 29 de [OMG09].

[2] Flatten est une opération sur une collection. Elle a pour effet qu'une collection ne contient plus d'autres collections comme éléments. Voir page 25 de [OMG09].

[3] Let : Parfois une sous-expression est utilisée plusieurs fois dans une expression. *let* permet.de déclarer et de définir la valeur (*i.e* initialiser) d'un attribut qui pourra être utilisé dans l'expression qui suit le *in*. La syntaxe de l'opérateur *let* est la suivante :
`Let <déclaration> = <requête> in <expression>`
voir page 11 de [OMG09].

L'étude comparative du cœur a été restreinte aux invariants car tous les moteurs OCL n'ont pas l'option d'évaluer les requêtes OCL, c'est-à-dire autre chose que des invariants.
En fait cela revient à comparer les requêtes avec les résultats attendus dans le but d'obtenir une expression booléenne.
Pas moins de six syntaxes différentes (qui sont décrites ci-après) sont fournies pour traiter des invariants.
Idéalement le parseur du moteur d'OCL accepte toutes les syntaxes, mais lors de notre étude nous considérerons que l'outil répond aux normes si au moins une syntaxe est acceptée par l'outil OCL. Trois choix voient le jour pour nommer et typer les variables dans les opérations sur les collections :

- Les variables itérateur peuvent être explicitement définies :`Person.allInstances()->reject(p|p.gender='male')`
- De plus elles peuvent être typées `Person.allInstances()->reject(p :Person|p.gender='male')`
- Et plusieurs opérations acceptent aussi les variables implicites : `Person.allInstances()->reject(gender='male')`

Le nombre de choix est doublé quand on incorpore la notation `allInstances()` sans parenthèses comme c'est permis dans la version 1.3 d'OCL
Après la vérification de syntaxe l'évaluation de la conformité est réalisée à l'aide d'un exemple de diagramme d'objet représentant une photo à un moment donné d'un état valide du système. Tous les invariants du cœur sont conçus pour être vérifiés dans le contexte de cet état du système.

- **B2 : les énumérations, les prés- et post conditions, les requêtes**

Durant l'étude du cœur, les éléments de base du modèle sont vérifiés, l'étude du cœur étendue ajoute les énumérations, les pré- et post-conditions et les requêtes dépendantes. Le focus sur les requêtes est l'accès aux objets (incluant le traitement des valeurs `undefined`) et la navigation ainsi que le traitement des littéraux d'énumération et les types d'attributs d'énumération comme le montre la figure suivante

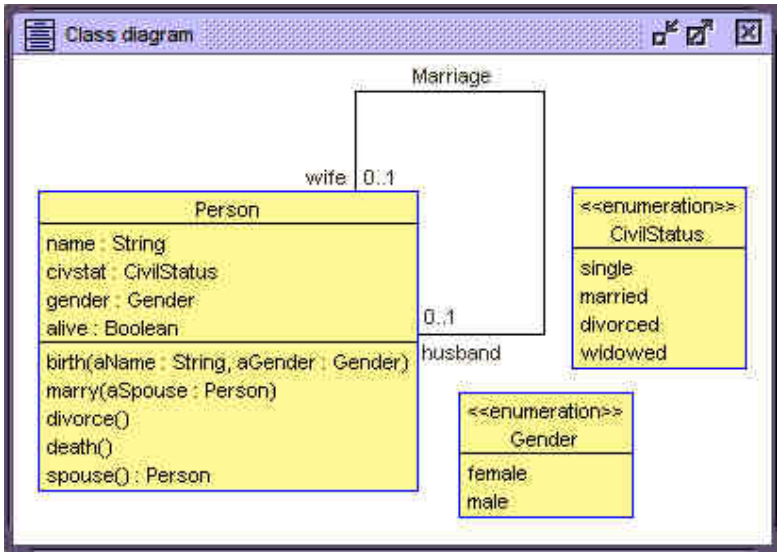


Fig. 10. Diagramme de classe du modèle ecore étendu

Dans ce scenario plusieurs diagrammes d'objet sont construits pour représenter un système en cours de d'exécution. Chaque paire d'états successifs représente l'exécution d'une opération spécifiée dans le modèle étendu.

- **B3 : les associations ternaires et les associations de classe**

Les associations ternaires ou plus et les classes-associations représentent un chapitre avancé dans la norme OCL [OMG09]. Les associations de ce type sont parfois nécessaires pour synthétiser le modèle et sont courantes dans la conception de modèles de bases de données.

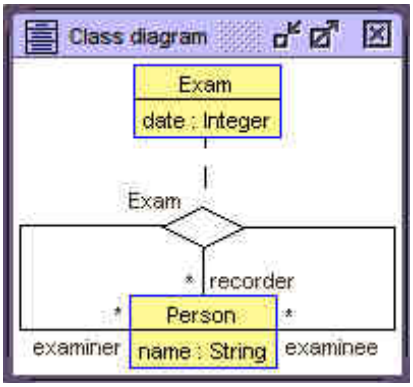


Fig. 11. Diagramme de classe du modèle avancé

Pour cette raison, la vérification de la conformité durant l'étude B3 est basée sur un modèle spécifiant une association de classe ternaire et réflexive. Un lien c'est-à-dire une instance de la classe-association Exam, est identifiée par un triplet de personnes. Chaque personne est

autorisée à participer à l'examen mais chacune a un rôle différent. L'expression suivante navigue à travers l'association ternaire.

```
let ada = Person.allInstances()->any(name='Ada') in
ada.examiner[recorder]
```

Les crochets indiquent le sens de navigation dans l'association. Pour cela l'expression du dessus a pour résultat l'ensemble des examinateurs étant présents à l'examen dans lequel Ada est le rapporteur (recorder en anglais). Par contre ada.examiner[examinee] a pour résultat toutes les personnes examinateurs à l'examen pour lequel Ada est l'examiné.

- B4 et B5 : les opérations et les propriétés appliquées aux collections
- B4 Les trois valeurs logiques de l'étude comparative.

OCL offre un traitement sophistiqué des « `undefined values` ». Ceci implique une logique à trois valeurs qui est testée dans la 4ème partie : B4 de l'étude de conformité. Si on veut respecter la sémantique définie dans la norme OCL [OMG09], B4 teste systématiquement l'implémentation correcte des opérations booléennes OCL dans un contexte de requêtes.
L'étude insiste sur le fait que le standard d'OCL requiert explicitement que par exemple « true **ou** n'importe quoi » est toujours vrai, et « false **et** n'importe quoi » est toujours faux.
Cela signifie que dans ces cas-là le « `undefined values` » ne se propage pas.

- B5 les normes OCL de l'étude comparative

L'étude B4 a pour objectif de vérifier systématiquement l'implémentation correcte d'opérations individuelles, avec un gros plan sur les opérations sur les collections. L'analyse des propriétés sémantiques des opérations OCL présentées en [GK07] fournit une base pour cette étude. Chaque test vérifie l'équivalence de deux expressions OCL entre elles et si chaque expression est vraie. S'il n'y a pas équivalence, au moins un cas d'erreur est signalé et décrit
. Les exemples suivants montrent une règle prise en compte dans l'étude comparative. La variable « e » représente une expression OCL booléenne.

```
let c = sourceCollection in c->exists(i|e) = c->select(i|e)->notEmpty()
```
[4]

[4] collection->select(v | boolean-expression-with-v) : Leparamètre select a une syntaxe particulière qui permet de spécifier quels éléments de la collection nous voulons sélectionner. La variable *v* est appelée l' iterateur. Quand le select est évalué, v itère la collection et *boolean-expression-with-v* est évalué pour chaque *v*. Le *v* est une référence à l'objet de la collection et peut être utilise pour faire référence aux objets de la collection

L'utilisation d'opérations générales comme « iterate » sur les collections pour remplacer d'autres opérations est un autre aspect important. Voici l'exemple ci-dessous :

```
let c = sourceCollection in c->exists(i|e) = c->iterate(i;r:Boolean=false|r
or e)
```
[5]

Pour vérifier une règle dans l'étude, nous devons remplacer l'expression correspondante par une collection source concrète : « c », et les expressions OCL : « e ». Dans le cas des expressions booléennes, une expression très simple comme « i<4 » est suffisante pour tester, car nous avons seulement besoin d'une expression qui a comme résultat vrai ou faux dépendante de la valeur de la variable de l'itérateur. L'évaluation d'une expression OCL trop compliquée peut être fausse. Cet aspect là est pris en compte dans d'autres parties de l'étude.
Au contraire, la collection source et son environnement doivent être systématiquement pris en compte, car non associée à son environnement l'évaluation de la collection peut être différente.
Par exemple :
inv r1 :let c:Set(Integer) =Set{} in c->reject(i|i<4) = c->select(i|not(i<4))
fonctionne car la collection choisie est un Set de Integer,
alors que
inv r2 : let c:Set(gens::Personne)= Set{} in c->reject(i|i<4) = c->select(i|not(i<4))
ne fonctionne pas car on ne peut pas appliquer des opérations comme i<4 si i est de type gens::Personne
Partant de ce fait, une règle est instanciée avec (1) des sets[6], bags[7] et sequences[8], (2) les collections vides, les collections avec un seul élément ou les collections avec beaucoup d'éléments, (3) les collections incluant et excluant une `valeur indéfinie`, (4) les collections incluant des éléments qui vérifient ou pas des expressions booléennes. Dans le cas des bags et des sequences nous faisons de plus la différence entre (5) les collections excluant ou pas les éléments égaux entre eux (6) qui vérifient ou pas les expressions booléennes. La combinaison de ces six situations a comme résultat vingt-neuf cas pour chaque équivalence. Voici un exemple ci-dessous :

---

[5] collection->iterate(elem : Type; acc : Type = <expression> | expression-with-elem-and-acc). La variable *elem* est l'iterateur, comme dans la définition de *select, forAll,* etc. La variable *acc* est l'accumulateur. L'accumulateur prend une valeur initiale <*expression*>. Quand l'iterate est évalué, *elem* itère sur la *collection* et *expression-with-elem-and-acc* est évalué pour chaque *elem*. Après chaque evaluation de *expression-with-elem-and-acc*, sa valeur est assignee à *acc*.
[6] Set : collection non ordonnée et sans doublons
[7] Bag : collection non ordonnée pouvant contenir des doublons
[8] Sequence : collection ordonnée pouvant contenir des doublons

```
let c = Set{-1,0,1,2}
in c->collect(i|i*i) = c->iterate(i;r:Bag(Integer)=Bag{}|r->including(i*i))
```

La partie B6 teste l'aspect déterminisme.

Cette partie de l'étude traite des propriétés d'implémentation du moteur OCL pour les caractéristiques non déterministes d'OCL et les opérations pour lesquelles le standard OCL autorise un choix dans l'implémentation comme `any` ou `flatten`. Le but de cette étude est de réduire la liberté de choix d'implémentation aussi loin que possible.

Dans OCL, il y a au moins cinq possibilités de convertir les sets et les bags en sequences. (Une séquence est une collection ordonnée qui peut contenir des doublons, alors qu'un set est une collection non ordonnée mais sans doublons).

Là nous allons seulement discuter des possibilités pour les sets parce que les conversions pour les bags sont analogues à celles des sets. En gros, les sets peuvent être changés en sequences en utilisant (1) `asSequence` , (2) `iterate`, (3) `any`, (4) `flatten` (une collection à laquelle on a appliqué l'opération « flatten » ne contient plus de sous-collection comme élément. elle est dite applatie).ou (5) `sortedBy`. Dans les expressions ci-dessous, `inSet` est une expression arbitraire OCL avec le type `Set(Integer)`.

Par exemple `Set{1..12}`

```
(1) intSet->asSequence()

(2) intSet->iterate(e:Integer;
       r:Sequence(Integer)=Sequence{}|
       r->including(e))

(3) intSet->iterate(u:Integer;
       r:TupleType(theSet:Set(Integer),theSeq:Sequence(Integer))=
         Tuple{theSet= intSet,theSeq=Sequence{}}|
       let e : Integer =r.theSet->any(true) in
           Tuple{theSet=r.theSet->excluding(e),
                 theSeq=r.theSeq->including(e)}).theSeq

(4) Sequence{intSet}->flatten()

(5) intSet->collectNested(e:Integer|Sequence{0,e})->
       sortedBy(s:Sequence(Integer)|s->first())->
       collect(s|s->last())
```

La première possibilité est la conversion directe avec `asSequence`. La seconde expression utilise un `iterate` sur l'ensemble composé d'entier avec une variable élément et construit successivement la séquence en ajoutant l'élément courant.

L'idée de base qui se cache derrière la troisième expression est de choisir un élément arbitraire avec `any` et d'ajouter cet élément à la séquence résultat.

La quatrième option fait appel à un `flatten` sur une séquence possédant un ensemble d'entiers comme seul élément

La cinquième expression utilise le `sortedBy` pour trier un bag d'entiers.

SortedBy appliqué à une collection a comme résultat un ensemble ordonné contenant tous les éléments de la collection source, ils sont alors classés en tenant compte du résultat de l'évaluation de l'expression passée en paramètre à l'opération.

Chacune des cinq solutions représente une manière particulière de produire une séquence à partir d'un set. Les résultats obtenus pourraient donc être légèrement différents d'où la notion de déterminisme.

Notre étude consiste à vérifier que l'ordre des éléments produit par les options (2) et (5) est le même que l'ordre produit directement par la commande `asSequence` (option 1).

En effet dans(3), un élément ayant pour valeur true est choisi arbitrairement dans le set avec any et ensuite il est exclu du set et mis dans la sequence.

Le cas (4) part d'une séquence qui peut contenir elle-même une collection à laquelle peut être optionnellement appliquée une opération, évalue cette opération grâce à l'opération flatten() et renvoie une Séquences d'éléments.

L'étude B6 vérifie aussi quelques aspects mineurs comme la fabrication d'un bag à partir d'un set.

```
aSet->any(true) = aSet->asBag()->any(true)
aSet->asSequence() = aSet->asBag()->asSequence()
```

Nous comprenons ces propriétés de déterminisme comme des aspects sous-jacents de la norme OCL car il n'est pas clairement explicité dans la norme quelle option a été choisie pour transformer un Set en Séquence : pour évaluer l'option (1) laquelle des options suivantes de (2) à (5) a été choisie ? Notre étude donne la possibilité de réduire cet aspect sous-jacent, ce « non-dit », et ce flou de libertés pour le développeur OCL.

La partie B7 teste l'efficacité d'OCL

Dans cette section, l'étude propose l'évaluation des expressions OCL vérifiant l'efficacité du moteur OCL. Les expressions sont supposées être évaluées dans les différents moteurs et le temps d'évaluation doit être enregistré. Dans le but d'évaluer facilement et correctement le temps d'exécution des expressions, les expressions seront évaluées plusieurs fois grâce à un iterate. L'évaluation des expressions est faite dans cette section d'une part sur un modèle contenant des booléen, des chaînes de caractères, des entiers et des réels, et d'autre part sur un petit modèle composé de villes et de leurs rues.

Les expressions pour le type de données traitent (A) la table de vérité des booléens disponibles dans OCL, (B) l'inversion d'une longue chaîne de caractères, (C) le calcul des nombres premiers jusqu'à une valeur maximale, (D) la racine carrée d'un nombre réel. Comme exemple, considérez l'exemple suivant d'une expression OCL pour les nombres premiers jusqu'à 2048.

```
Sequence{1..2048}->iterate(i:Integer;
   res:Sequence(Integer)=Sequence{}|
      if m.isPrime(i) then res->including(i) else res endif)

isPrime(arg:Integer):Boolean=
   if arg<=1 then false else
      if arg=2 then true else isPrimeAux(arg,2,arg div 2) endif endif
isPrimeAux(arg:Integer,cur:Integer,top:Integer):Boolean=
   if arg.mod(cur)=0 then false else
      if cur+1<=top then isPrimeAux(arg,cur+1,top) else true endif
   endif -- algorithm inefficiency irrelevant for benchmark
```

Les expressions pour le modèle exemple avec les villes et les rues prennent en compte la structure des données sous-jacentes comme un graphe avec des objets (les nœuds) et des liens (les arcs). Ils traitent automatiquement (A) les fermetures transitives, c'est-à-dire les nœuds accessibles directement ou indirectement à partir d'un nœud donné et (B) les composants reliés entre eux sur un graphe c'est-à-dire les ensembles contenant un maximum de nœuds dans lesquels les nœuds sont reliés directement ou indirectement entre eux c'est-à-dire : la connectivité du graphe. Le modèle exemple est constitué d'une seule classe et une seule association comme affiché à la figure 5.

Un exemple d'état : [GKB08a] page 238, avec 42 villes et 42 rues est ainsi construit.

Le graphe sous-jacent a 5 composants connectés avec les nœuds numéros 1, 2, 3, 13 et 23.

Dans cet exemple l'expression OCL suivante pour la fermeture transitive est évaluée.

```
Set{1..1024*1024}->iterate(i:Integer;
  res:Bag(Set(Town))=Town.allInstances->collect(t|t.connectPlus())|res)
```

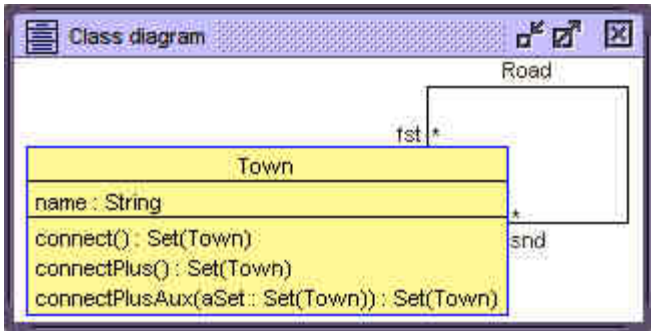


Fig. 12. Diagramme de classe des villes et des rues

L'opération `connectPlus()` calcule automatiquement toutes les villes directement ou indirectement accessibles à partir du noeud courant les rôles fst pour first (ville connectée directement à la ville initiale) et snd pour second (ville connectée à une autre ville directement connectée à la ville initiale).
Tous les détails, c'est-à-dire les modèles, les invariants et les requêtes de cette partie et des autres parties de l'étude comparative d'OCL se trouvent dans [GKB08a].

DEMARCHE A SUIVRE

Dans chaque groupe, M. Gogolla [GKB08b] propose deux types de tests ***:***

- **Ceux concernant la bonne syntaxe de l'expression OCL (sa grammaire)**

  Ex : allInstances avec ou sans parenthèses : allInstances() ou allInstances

- **Et ceux dont la syntaxe est correcte mais dont les résultats attendus peuvent varier selon les interprètes utilisés**

Par exemple, le résultat de l'expression Set{1,2,3}→ collectNested(i | Sequence{i,i*i}) peut fournir comme résultat les trois valeurs suivantes :

- Bag{Sequence{1,1}, Sequence{2,4}, Sequence{3,9}}
- Bag{Sequence{2,4}, Sequence{1,1}, Sequence{3,9}}
- Bag{1,1,2,4,3,9}

Toutefois, si on se réfère à la norme, seule la première réponse est correcte (voir page 29 de [OMG09])

Les tests concernant la syntaxe seront effectués automatiquement par les outils à la compilation ils ne feront donc pas partie de notre étude. Nous nous focaliserons donc uniquement sur la vérification des résultats.

Une des premières pistes pour tester la validité de la syntaxe est de commencer par étudier ce qui se passe dans le cas du traitement des invariants c'est-à-dire en suivant la chronologie de l'article de M. Gogolla contenant les cas de test [GKB08a]. Il faudra aussi étudier que tous les cas posant problème ont bien été balayés.

En s'inspirant de la norme OCL [OMG09], et des « *Well-formedness Rules* », on peut dresser une première liste des types d'opérations à tester :

- Les réels, les entiers, les chaines de caractère, les booléens voir la norme [OMG09], page 141, ceci est traité dans B1 de [GKB08a] et [GKB08b]
- Les expressions logiques : les invariants, les prés et post conditions voir page 159 de la norme [OMG09] sont traités dans B1 de [GKB08a] et [GKB08b]
- Les collections et leurs opérations, voir page 145 de la norme [OMG09] sont traités dans B4 et B5 de [GKB08a] et [GKB08b]
  - Les opérations sur les collections
  - Les opérations sur les Set
  - Les opérations sur les ensembles ordonnés (orderSet)
  - Les opérations sur les sacs (Bag)
  - Les opérations sur les sequences (Sequence)
- Les itérateurs sur les collections, voir page 155 de la norme [OMG09], ceci est traité dans B1, B2, B5, B6, B7 de [GKB08a] et [GKB08b]
- Les types OclAny, OclVoid, OclInvalid, and OclMessage, qui constituent les types de la bibliothèque standard d'OCL voir page 137 de la norme [OMG09].

La syntaxe des types de la bibliothèque standard d'OCL :
Cette partie décrit les bibliothèques standards d'OCL des types prédéfinis, leurs opérations, et les schémas d'expressions prédéfinies sur ces types. Pour chaque opération la signature et la description de la sémantique est donnée.
OCLAny, OCLVoid, OCLInvalid, et OCLMessage : voir p 139-140 de [OMG09]

- OclAny : Toutes les classes dans un modèle UML héritent des opérations définies avec OclAny. Pour éviter des conflits de noms entre les propriétés dans le modèle et les propriétés héritées d'OclAny, tous les noms dans les propriétés d'OclAny commencent par « ocl ». ceci est traité dans B2 de [GKB08a] et [GKB08b]

- OclVoid est un type qui est conforme à tous les autres types sauf OclInvalid, il a une unique instance, identifiée à « null » qui correspond à la valeur de la spécification UML « LiteralNull ». N'importe quel appel de propriétés appliquée à « null » a comme résultat OclInvalid excepté pour l'opération OclIsUndefined() et OclIsInvalid(). Cependant, dans un

souci de cohérence avec la conversion implicite à un littéral d'une collection, une expression évaluée à null peut être utilisée comme source d'une opération sur une collection (comme isEmpty). Si la source est le littéral null, elle est implicitement convertie en Bag{}. OclVoid est lui-même une instance du méta type VoidType. Ce cas n'est pas traité dans [GKB08a] et [GKB08b].

- OclInvalid est un type qui se conforme à tous les autres types sauf OclVoid, il a une unique instance, identifiée à « invalid ». N'importe quel appel de propriété appliqué à « invalid » a pour résultat OclInvalid, excepté pour l'opération OclIsUndefined () et OclIsInvalid(). OclIsInvalid() est lui-même une instance du méta type InvalidType. Ce cas n'est pas traité dans [GKB08a] et [GKB08b]

- OclMessage .On peut trouver des messages dans 3 types d'opérations :
  - hasReturned() : Boolean
    retourne vrai si le paramètre du modèle est une opération call, et que l'opération call a retourné une valeur. Cela implique que le message a été envoyé.
    Retourne faux dans tous les autres cas
  - isSignalSent() : Boolean
    retourne vrai si OclMessage représente l'envoie d'un signal UML.
  - isOperationCall() : Boolean
    retourne vrai si OclMessage représente l'envoie d'une opération UML call
    voir [OMG09] page 141.

Cette partie contient la définition du type standard OclMessage. Chaque message Ocl est actuellement un type de message avec un paramètre T. Un message concret Ocl est créé en remplaçant une opération ou un signal par T. Le type prédéfini OclMessage est une instance de MessageType. Chaque OclMessage est entièrement déterminé par soit une opération soit un signal donné par un paramètre. Ce cas n'est pas traité dans [GKB08a] et [GKB08b]

En considérant la norme OCL, on s'aperçoit qu'elle traite essentiellement de la syntaxe des expressions OCL ce qui est le rôle du compilateur ou d'un interprète OCL. Nous allons en plus nous intéresser au type du résultat et à la valeur du résultat qu'on attend.

Pour informatiser ces tests, on utilisera des requêtes OCL de type invariant comparant le résultat d'une expression et sa réponse théorique pour la validité du résultat, et l'utilisation de

l'instruction *oclIstypeof* pour tester le type du résultat. La propriété *oclIsKindOf* détermine si *t* est du type ou d'un sous-type passé en paramètre.

Par exemple

Inv r31:
mmCec::Cec::Personne.allInstances()-
>collect(name)=Bag{"Tata,"Ada","Dan","Cyd","Bob")

Inv r32:
mmCec::Cec::Personne.allInstances()->collect(name)
->oclIsTypeOf(Bag(String))

Adaptation des règles pour les différents outils

Bien que les outils considérés dans cette étude supportent la norme OCL, il existe quelques variations syntaxiques qui nécessitent une adaptation des règles pour chaque outil.

Considérons la règle OCL suivante vérifiant qu'une personne est soit seule, mariée, divorcée ou veuve :

> *Context person*
> *inv enumCivilStatus :*
> *Self.civstat='single' or self.civstat='married' or self.civstat='divorced' or self.civstat='widowed',*

Dans le cas de l'outil Neptune il faut apporter quelques précisions à la définition du contexte. En effet, pour que la règle soit correctement exécutée, il faut que le contexte contienne le nom du métamodèle, suivi du nom du modèle et du nom de la classe les trois séparés par un double deux points. Par exemple, la règle précédente s'écrira pour NEPTUNE de la manière suivante :

```
Context mmCec[monModele]::Cec::Personne
Inv enumCivilStatus :
Self.civstat=Civstatus::single or Self.civstat=Civstatus::married or
Self.civstat=Civstatus::divorced or Self.civstat=Civstatus::widowed
```

Avec *mmCec* le nom du métamodèle considéré, *monModele* le modèle considéré et *Cec* le paquetage dans lequel se trouve la classe *Personne*.

Dans le cas de l'outil Topcased le contexte ne doit contenir que le nom du paquetage contenu dans le modèle ecore ici *gens,* suivi d'un double deux points, suivi du nom de la classe ici *Personne*. Il faut écrire:

```
Context gens::Personne
Inv enumCivilStatus :
```

```
Self.civstat=Civstatus::single or Self.civstat=Civstatus::married or
Self.civstat=Civstatus::divorced or Self.civstat=Civstatus::widowed
```

Dans Neptune isEmpty sans les parenthèses fonctionne.
Dans Topcased isEmpty sans les parenthèses ne fonctionne pas. Il faut l'écrire isEmpty().dans la norme isEmpty est toujours suivi de parenthèses alors que dans l'article de M Gogolla [GKB08a] isEmpty est sans parenthèses voir page 37. La syntaxe sans parenthèse provient des anciennes versions de la norme OCL.

De plus Neptune et Topcased acceptent tous les deux les expressions suivantes pour les opérations d'itération sur les collections :

- collect(p|p.name)
- collect(p :Personne|p.name)
- collect(name)

Pour tester des collections de collections il faut utiliser collectNested. En effet dans OCL 1.4 les collections sont toujours « flatten » c'est-à-dire qu'une collection ne peut pas contenir une sous-collection comme élément. Cette restriction est levée dans OCL 2.0, OCL permet maintenant que les éléments d'une collection soient elles-mêmes des collections. La bibliothèque standard d'OCL inclut des opérations spécifiques qui permettent de « calculer les opérations » faites sur des sous-collections de collections Celles-ci peuvent être utilisées pour aplatir explicitement les collections de collections. Voir page 25 de la norme OCL [OMG09].

Considérons l'instruction suivante :

```
source->collect (iterator | body) = source->collectNested (iterator |
body)->flatten()
```

L'instruction « collect » a pour résultat la collection d'éléments qui résulte de l'application de « body » à l'ensemble « source ». Le résultat est aplati. Ceci ne peut se faire que grâce à l'utilisation de collectNested, qui peut être de différents types dépendants du type de « source ». « collectNested » est défini individuellement pour chaque type de collection (c'est-à-dire les Set, Bag ,orderedSet, Sequence). Voir « collect » page 163 de [OMG09]

Dans la partie suivante, nous allons illustrer au travers d'un exemple les tests effectués sur des collections OCL car l'étude de M.Gogolla [GKB08a] commence par traiter des invariants avec des collections

Les collections

Considérons l'exemple présent dans l'article de M. Gogolla et qui manipule un ensemble, au sens mathématique, de personnes. Nous désirons extraire de cet ensemble le nom des personnes contenues dans cet ensemble. Le Résultat attendu est le suivant :

Bag{'Ada','Bob','Cyd','Dan','tata'} de type Bag(String)

La règle permettant de vérifier propriété est la suivante :

Context Person

Inv : Personne.allInstances()-> collect (name) =Bag{'Ada','Bob','Cyd','Dan','tata'}
Inv : Personne.allInstances()->collect(name)->oclIsTypeOf(Bag(String))

Le premier invariant teste que le résultat retourné par l'évaluation est correct, le second teste que le type est correct. Comme pour tout invariant les réponses fournies seront de type booléen et si la mise en œuvre de l'interprète OCL est correcte ces deux invariants retourneront comme résultat vrai.

Lors de l'évaluation de cette règle dans Topcased, **le premier invariant** retourne vrai

```
inv r32:
Personne.allInstances()->collect(name)=Bag{'Ada','Bob','Cyd','Dan','tata'}
```

Dans neptune : même résultat

Le second invariant

```
inv r33:
Personne.allInstances()->collect(name)->oclIsTypeOf(Bag(String))
```

n'est pas accepté par l'interprète OCL de Topcased, (alors qu'il est accepté dans Neptune), le message d'erreur est le suivant :

Parsing Exception : Unrecognized variable: (Bag) -- error location not published by the parser

il faudra le transformer en deux invariants pour qu'il soit évaluable par Topcased:

```
context gens::Personne
inv r35 :
```

```
Personne.allInstances()->collectNested(name)->oclIsTypeOf(Bag)
```

Qui génère une erreur :

Parsing Exception : 10:59:10:61 "Bag" unexpected token(s)

La solution est de soumettre l'invariant suivant

```
inv r32:
Personne.allInstances()->collect(name)=Bag{'Ada','Bob','Cyd','Dan','tata'}
```

Ici on voit bien que le résultat est un Bag{String}

Et

```
inv r34 :
Personne.allInstances()->collectNested(name)->
forAll(s|s.oclIsTypeOf(String))
```

qui fonctionne.

Complétons l'exemple traité précédemment afin d'extraire de cet ensemble une séquence constituée du nom, de son statut (mariée, veuve, divorcée, célibataire), de son sexe et de si elle est vivante ou pas. Le Résultat attendu est le suivant :

Bag{ Sequence{'Ada', Civstatus::widowed, Gender::female,true },
Sequence{'Bob', Civstatus::divorced, Gender::male, true },
Sequence{'Cyd', Civstatus::married, Gender::male, false},
Sequence{'Dan', Civstatus::single, Gender::male, true }}
de type Bag(Sequence(OclAny))

La règle permettant de vérifier propriété est la suivante :

Context Person
Inv: Person.allInstances-> collectNested(Sequence{name,civstat,gender,alive})=
Bag{Sequence{'Ada', Civstatus::widowed, Gender::female,true },
Sequence{'Bob', Civstatus::divorced, Gender::male, true },
Sequence{'Cyd', Civstatus::married, Gender::male, false},
Sequence{'Dan', Civstatus::single, Gender::male, true }}
Inv : Person.allInstances->collectNested(Sequence{name,civstat,gender,alive})->
oclIsTypeOf(Bag(Sequence(OclAny)))

Lors de l'évaluation de cette règle dans Topcased, l'invariant :

context gens::Personne
inv r14 : Personne.allInstances()-> collectNested(Sequence{name,civstat,gender,alive})=
Bag{Sequence{'Ada',Civstatus::married,Gender::female,true },
Sequence{'Bob',Civstatus::divorced,Gender::male, true },
Sequence{'Cyd',Civstatus::married, Gender::male, true},
Sequence{'Dan',Civstatus::single, Gender::male, true},
Sequence{'tata',Civstatus::married,Gender::female, true }}

A pour réponse vrai, ce qui est la réponse attendue

Il en est de même dans l'environnement NEPTUNE ce qui montre que les deux interprètes ont dans ce cas un comportement conforme à la norme.

Conclusion sur les invariants.

Dans le cas des requêtes, il faudra créer un invariant en notifiant le contexte :

Dans Topcased

```
context gens::Personne
```

C'est-à-dire context suivi de Nom du package ::nom de la classe

Dans Neptune

```
context mmCec[nomModele]::Cec::Personne
```

C'est-à-dire context suivi de nom du métamodèle, du nom du modèle entre crochets, du paquetage et la classe considérée.

Une fois notifié le contexte, il faudra créer un invariant avec le mot clé `inv` suivi du nom de l'invariant suivi de deux points c'est-à-dire inv nom_de_l'invariant :

```
inv enumCivilStatus :
```

Une fois l'invariant créé on rajoutera à la suite la requêtes de M Gogolla sans le point d'interrogation et on y ajoutera le signe = suivi du résultat attendu. Voici un exemple pour illustrer notre propos.

Soit la requête de création d'un nouvel attribut en utilisant l'instruction OCL « let ». Dans notre exemple, cet attribut contiendra l'instance de type Personne représentant la personne qui se prénomme « Ada ».

?let o:OclAny=ada in o

Résultat attendu:
ada : Person

La syntaxe du let est la suivante :

Let <déclaration>=<requête> in <expression>

Un nouvel attribut déclaré dans **<déclaration>** aura comme valeur le résultat de l'expression **<requête>** durant l'évaluation de l'expression **<expression>**

Autrement dit dans notre exemple un nouvel attribut déclaré dans déclaration ici o aura comme valeur « ada » dans toute l'expression « o = 'Ada' :

Le résulta est donc « ada » qui est de type « Personne »

On écrira donc l'invariant :

```
inv r35:
let o:OclAny='Ada' in o='Ada'
```

Au passage voici une remarque sur la syntaxe du let suivant les deux outils Neptune et Topcased :

Considérons un invariant qui pour son évaluation nécessite la déclaration d'un attribut tel que présenté ci-dessus.

Avec l'environnement **Neptune**, nous écrirons :

```
context mmCec::Cec::Personne
inv r36 :
let o=mmCec::Cec::Personne.allInstances()->any(name='Ada') in
o.oclIsTypeOf(mmCec::Cec::Personne).
```

Par contre dans l'environnement **Topcased**, nous devrons écrire :

```
context gens::Personne
inv r36 :
let o:gens::Personne=Personne.allInstances()->any(name='Ada') in
o.oclIsTypeOf(Personne)
```

Il faudra donc dans cette version spécifier le type de l'attribut lors de la déclaration de « o ». Cette différence de syntaxe est due à une ambigüité de la norme OCL. En effet, si nous nous référons à la norme page 11 nous constatons que le type de o doit être précisé. Toutefois, si nous étudions la syntaxe concrète de la grammaire OCL pages 78 et 89, et en particulier la syntaxe d'une déclaration de variable (variableDeclaration (' :' typeCS) ?)) Nous constatons que le point d'interrogation signifie que la déclaration du type est optionnelle. L'environnement NEPTUNE supportant les deux alternatives nous utiliserons donc celle commune aux deux environnements.

Dans la partie suivante nous allons nous intéresser aux iterateurs qui constituent la base pour manipuler des collections.

Les itérateurs

Les opérations reject[9], forAll[10], exist[11], peuvent toutes être décrites sous la forme d'iterate.

Une accumulation de valeurs se construit en évaluant une expression sur une collection**.**

De façon générale une itération est de la forme :

`collection->iterate( elem : Type; acc : Type = <expression> | expression-with-elem-and-acc )`

La variable ***elem*** est l'itérateur, comme dans la définition du *select, forAll,* etc. La variable ***acc*** est l'accumulateur. L'accumulateur prend la valeur initiale <*expression*>. Quand l'iterate est évalué, *elem* itère sur la collection et ***expression-with-elem-and-acc*** est evaluée pour chaque ***elem*.** Après chaque évaluation de ***expression-with-elem-and-acc*** sa valeur est assignée à ***acc*.** De cette façon, la valeur de ***acc*** est construite durant l'itération de la collection.

Considérons la requête

```
Person::Person.allInstances()->
iterate(w,h: Person::Person;
        res:Bag(Sequence(Person::Person))=Bag{}|
         if w.gender=Gender::female and w.alive and
        w.civstat=Civstatus::maried and
              h.gender= Gender::male and h.alive and h.civstat =
              Civstatus::maried then
              res->including(Sequence{w,h})
         else res endif)->
collectNested(pair:Sequence(Person::Person)|pair->collectNested(name))
```

[9] Reject L'opération reject est identique à l'opération select, mais avec reject nous prenons le sous-ensemble de tous les éléments de la collection pour lesquels l'expression est évaluée à faux.
Select : select est une opération sur une collection et elle est spécifiée en utilisant une syntaxe avec une flèche: collection->select(...).
Le paramètre du select a une syntaxe spécifique qui permet de spécifier quel élément de la collection nous voulons sélectionner.

[10] forAll : Souvent une constraint est nécessaire pour tous les elements d'une collection. L'opération forAll en OCL permet de spécifier une expression booléenne, qui doit être vérifiée pour tous les éléments d'une collection: collection->forAll(v : Type | boolean-expression-with-v)
L'expression *forAll* a comme résultat un booléen. Le résultat est true si *boolean-expression-with-v* est true pour tous les éléments de la collection. Si t *boolean-expression-with-v* est false pour un ou plusieurs *v* dans une collection, alors toute l'expression est évaluée à false

[11] Exist : Souvent on a besoin de savoir si il y a au moins un élément de la collection pour lequel la contraint est vérifiée. L'opération *exists* dans OCL vous permet de spécifier une expression booléenne qui doit être vérifiée pour au moins un élément dans la collection

Dans notre cas **w** et **h** sont les itérateurs et **res** est l'accumulateur. Initialement, il est vide. Il est construit progressivement au cours des évaluations successives. A la fin, il contient toutes les personnes qui sont mariées.

Lors de l'évaluation de cette règle dans l'environnement NEPTUNE

```
context mmCec::Cec::Personne
inv in10 :
 mmCec::Cec::Personne.allInstances()->iterate(w,h:mmCec::Cec::Personne
;res:Bag(Sequence(mmCec::Cec::Personne))=Bag{}|
   if w.gender=Gender::female and w.alive and w.civstat= Civstatus::maried
and
    h.gender= Gender::male and h.alive and h.civstat = Civstatus::maried
    then res->including(Sequence{w,h})
    else
    res endif)->collectNested(pair:Sequence(mmCec::Cec::Personne)|pair->
collectNested(name))=Bag{Sequence{'Ada','Cyd' }}
```

On a rajouté "`context mmCec::Cec::Personne` " et "`inv :` » devant la requête comme c'est une collection on a rajouté « = » après la requête suivi de la réponse attendue

Pour finir, il faut tester que le résultat est un Bag(Sequence(mmCec::Cec::Personne))

Pour cela nous complétons l'invariant que nous venons de tester afin de tester le type.

```
context mmCec::Cec::Personne
inv in10 :
 mmCec::Cec::Personne.allInstances()->iterate(w,h:mmCec::Cec::Personne
;res:Bag(Sequence(mmCec::Cec::Personne))=Bag{}|
   if w.gender=Gender::female and w.alive and w.civstat= Civstatus::maried
and
    h.gender= Gender::male and h.alive and h.civstat = Civstatus::maried
    then res->including(Sequence{w,h})
    else
  res
 endif)->collectNested(pair:Sequence(mmCec::Cec::Personne)
|pair->collectNested(name))    ->oclIsTypeOf(Bag(Sequence(String)))
```

L'expression ci-dessus est parfaitement conforme à la syntaxe OCL. Malheureusement, elle n'est pas interprétée par l'environnement TOPCASED car l' « iterate » de l'expression nécessite deux variables d'itération w et h alors que dans Topcased une seule valeur d'itération est autorisée. La norme semble indiquer page 31 que l'iterate ne supporte qu'une variable d'iteration. Toutefois, les opérations comme forAll peuvent être décrites sous la forme d'un iterate et un exemple d'utilisation du « forAll » page 30 montre qu'il peut avoir deux variables d'iteration. De plus, la syntaxe concrète d'OCL page 76 semble confirmer la possibilité d'avoir deux variables d'itération. Il résulte de ce fait que la règle précédemment

mentionnée n'est pas exécutable dans l'environnement TOPCASED ce qui semble montrer une violation de la norme.

J'ai essayé l'invariant suivant :

```
context gens::Personne
inv in10 :
 gens::Personne.allInstances()->iterate(w:gens::Personne ;h:gens::Personne
;res:Bag(Sequence(gens::Personne))=Bag{}|
   if w.gender=Gender::female and w.alive and w.civstat= Civstatus::married
and
    h.gender= Gender::male and h.alive and h.civstat = Civstatus::married
    then res->including(Sequence{w,h})
    else
    res
endif)->collectNested(pair:Sequence(gens::Personne)|pair-
>collectNested(name))=Bag{Sequence{'Ada','Cyd' }}
```

et j'ai l'erreur

Parsing Exception : 18:57:18:73 ",h:gens::Personne" misplaced construct(s)

Conclusion : En ce qui concerne la syntaxe de « iterate », TOPCASED ne respecte pas la norme car la syntaxe normale (voir ci-dessus) n'est pas acceptée par le parseur de TOPCASED.

Dans Topcased l'invariant:

```
inv r40 : true or Sequence{true}->excluding(true)->last()
```

est vrai

explication:
Sequence{true} est une suite ordonnée de « true » limité à un élément
excluding(true) a pour effet de supprimer les valeurs true dans la sequence, la liste est alors vide.
last() prend le dernier élément qui est OclUndefined ensuite il évalue true or OclUndefined et le résultat est true

Dans Neptune le même invariant est vrai ce qui est correct.

Dans Neptune, la requête :

```
query:
let B=Set{Sequence{true}->excluding(true)->last(),false,true} in
B->iterate(b1,b2:  Boolean;r:Sequence(Boolean)=Sequence{}|  r->including(b1
or b2))
```

A comme résultat :

```
Sequence{false,true,Undefined,true,true,Undefined,Undefined,true,Undefined}
```

Alors que la réponse de M. Gogolla est

```
Sequence{ Undefined, Undefined,true,Undefined,false,true,true,true,true}
```

Or Set{Sequence{true}->excluding(true)->last(),false,true} représente le

set{OclUndefined,false,true}

Quand on évalue B->iterate(b1,b2: Boolean;r:Sequence(Boolean)=Sequence{}| r-> including(b1 or b2))
on obtient : les couples (b1,b2) suivants :

(OclUndefined,false) : OclUndefined or false a comme résultat **OclUndefined**
(OclUndefined,true) : OclUndefined or true a comme résultat **true**
(false,true) : false or true a comme résultat **true**
(false, OclUndefined) : false ou OclUndefined a comme résultat **OclUndefined**
(true, OclUndefined) : true,or OclUndefined a comme résultat **true**
(true,false) : true or false a comme résultat **true**
(OclUndefined, OclUndefined): OclUndefined or OclUndefined a comme résultat **OclUndefined**
(false,false) : false or false a comme résultat **false**
(true,true): true or true a comme résultat **true**

Donc l'invariant :

```
Inv :
let B=Set{Sequence{true}->excluding(true)->last(),false,true} in
B->iterate(b1,b2:  Boolean;r:Sequence(Boolean)=Sequence{}|  r->including(b1
or b2))
= Sequence{ Undefined, Undefined, true,Undefined,false,true,true,true,true}
```

Est faux dans Neptune

Etudions le cas dans Topcased:

```
inv r41 :
let B=Set{Sequence{true}->excluding(true)->last(),false,true} in B->
iterate(b1,b2:Boolean)=Sequence{}|r->including(b1 or
b2))=Sequence{Undefined,Undefined,true,Undefined,false,true,true,true,true}
```

ne passe pas à la compilation, le message est le suivant :

« let » unexpected token(s)

Si on ecrit :

```
inv r41 : let b:Sequence{Boolean}=Set{Sequence{true}->excluding(true)->
last(),false,true} in b->iterate(b1,b2:Boolean)=Sequence{}|r->including(b1
or
b2))=Sequence{Undefined,Undefined,true,Undefined,false,true,true,true,true}
```

C'est-à-dire en rajoutant le type de b, il s'arrête aussi à la compilation.

Autre invariant testé dans Topcased

```
inv toto:
let c:Integer=Bag{OclUndefined(Integer)} in c->reject(i|i<4)=c->
select(i|not(i<4))=true
```

cet invariant ne compile pas, le message est « illegal OclUndefined(Integer) operation.

C'est normal car OclUndefined(Integer) n'existe pas dans la norme

Si on teste

```
inv toto:
let c:Integer=Bag{null} in c->reject(i|i<4)=c->select(i|not(i<4))=true
```

cet invariant ne compile pas, le message est « illegal OclUndefined(Integer) operation

Si on teste

```
inv toto:
let c:Integer=Bag{} in c->reject(i|i<4)=c->select(i|not(i<4))=true
```

la compilation passe mais l'invariant est faux. Ce résultat n'est en aucun cas conforme à la norme car une erreur de typage doit au moins retourner la valeur « invalid ». De plus, le risque avec un tel résultat est de fourvoyer le concepteur de la règle en retournant une valeur dont l'interprétation semble indiquer une incohérence dans le modèle considéré et non une erreur dans l'écriture de celle-ci.

C'est donc **une erreur de typage**, ici, si on définit comme type un mauvais type dans TOPCASED, il passe à la compilation alors qu'il devrait y avoir une erreur, il devrait nous demander de mettre pour c le type `Bag(Integer) à la place de Integer.`

```
let c:Bag(Integer)=Bag{}
```

il faut écrire :

```
inv toto:
let c:Bag(Integer)=Bag{} in c->reject(i|i<4)=c->select(i|not(i<4))=true
```

Ainsi, la compilation passe et l'invariant est vrai

Testons-le dans Neptune :

```
inv toto:
```

```
(let c:Bag(Integer)=Bag{} in c->reject(i|i<4)=c->select(i|not(i<4)))=true
```

Ainsi, la compilation passe et l'invariant est vrai.

Autre point remarqué :

Dans Topcased, l'instruction

```
inv r701:let c : Set(Integer) = Set{} in c->asSequence() =
c->iterate(u;
r:Tuple(theSet:Set(Integer), theSeq:Sequence(Integer)) =
Tuple{theSet:Set(Integer)=c,theSeq:Sequence(Integer)=Sequence{}} |
let e:Set(Integer) = r.theSet->any(true) in
Tuple{theSet:Set(Integer)=r.theSet->excluding(e),
theSeq:Sequence(Integer)=r.theSeq->including(e)}).theSeq
```

ne compile pas alors que dans Neptune si.

Dans Topcased on a le message suivant :

Parsing Exception : Cannot find operation (excluding(Set(Integer))) for the type (Set(Integer)) -- error location not published by the parser.

Il s'agit d'une erreur de typage dans Neptune car la norme dit :

```
excluding(object : T) : Set(T)
 The set containing all elements of self without object.
post: result->forAll(elem | self->includes(elem) and (elem <> object))
 post: self- >forAll(elem | result->includes(elem) = (object <> elem))
 post: result->excludes(object)
```

De plus Neptune impose l'utilisation de TupleType alors que Topcased non.

Dans Neptune,il faut écrire

```
inv r701:let c : Set(Integer) = Set{} in c->asSequence() =
c->iterate(u;
r:TupleType(theSet:Set(Integer), theSeq:Sequence(Integer)) =
Tuple{theSet:Set(Integer)=c,theSeq:Sequence(Integer)=Sequence{}} |
let e:Set(Integer) = r.theSet->any(true) in
Tuple{theSet:Set(Integer)=r.theSet->excluding(e),
theSeq:Sequence(Integer)=r.theSeq->including(e)}).theSeq
```

voir la norme page 27, il faut utiliser TupleType.

Maintenant que nous avons examiné comment tester les règles présentées dans l'article de M. Gogolla, nous allons vérifier les différents points de la norme OCL qui ont été présentés dans l'article de M. Gogolla et parallèlement nous allons mettre en œuvre l'outil de test choisi pour les automatiser.

Pour étudier si tous les points de la norme ont été vérifiés, nous balayons les différents cas de tests de l'article et nous regardons quelle partie de la norme OCL est impactée. Pour ce faire j'ai réalisé un tableau contenant la table des matières de la norme OCL dans une colonne et dans l'autre une croix si l'item de la table des matières de la norme est testé.
Dans un premier temps je vais faire l'inventaire de toutes les notions OCL abordées par l'article de M. Gogolla, je les reporterai dans une feuille d'un fichier EXCEL. Je ferai ensuite, dans une nouvelle feuille Excel, le même travail de collecte pour les notions abordées dans la norme OCL ; Pour finir je croiserai les deux feuilles pour identifier les notions qui n'ont pas été traitées.

Ayant étudié OCL et notamment les cas qui posent problème et maitrisant maintenant la partie technique des tests OCL dans NEPTUNE et TOPCASED, je peux aborder la partie automatisation.

SOLUTION D'INFORMATISATION

Dans TOPCASED nous observerons que le fichier résultat est un fichier en format XMI qu'il faudra surement convertir pour traiter les résultats qu'il contient.

Nous avons observé que pour utiliser Salome TMF nous avons besoin de lancer une requête qui peut être l'exécution d'une requête OCL dans NEPTUNE ou dans TOPCASED. Or Salome TMF ne permet pas de tester les applications utilisant des interfaces graphiques telles que celles présentes dans TOPCASED et NEPTUNE. Nous avons donc pris l'option de lancer en batch les requêtes OCL dans NEPTUNE et TOPCASED depuis Salome TMF. Pour NEPTUNE, pas de problème, nous disposons d'un programme Java qui permet de lancer en batch les requêtes OCL. Il est important de souligner qu'il n'existe aucune documentation sur TOPCASED expliquant la procédure complète à suivre pour créer un métamodèle (fichier .xmi ou .ecore), générer le code java pour manipuler ce métamodèle, créer des modèles conformes à ce métamodèle et exécuter des contraintes OCL. En particulier la documentation est inexistante pour exécuter les contraintes en mode batch. Ma première tâche a donc été d'effectuer des recherches sur Internet et de réaliser de nombreuses manipulations, souvent en tâtonnant, afin d'appréhender et de maîtriser toute la chaîne TOPCASED allant de la création des métamodèles à l'exécution en batch des requêtes OCL.

J'ai profité de cet apprentissage pour réaliser une documentation qui résume les différentes étapes à suivre pour mettre en œuvre sous TOPCASED cette chaîne de production.

Une fois l'utilisation de TOPCASED pour exécuter des requêtes OCL maîtrisée, j'ai réfléchi à comment automatiser les campagnes de tests permettant de vérifier la conformité des interprètes OCL considérés et en particulier comment automatiser le benchmark de M. Gogolla qui sert de base à notre étude. Il m'a donc fallu à partir de la réponse théorique calculée manuellement ou proposée dans le document de M. Gogolla, tester si celle-ci était bien conforme à la réponse obtenue par les différents outils. Comme nous l'avons présenté précédemment ceci est rendu possible par l'utilisation d'invariants en faisant en sorte que si le teste est probant l'évaluation de l'invariant retourne toujours « vrai ».

Nous avons donc en entrée de NEPTUNE et TOPCASED un fichier contenant des invariants dont la réponse attendue doit être vrai.

Dans NEPTUNE nous observerons que le fichier résultat est un fichier texte, qui contient sur une ligne la requête sous forme d'invariant, le type du résultat (Boolean) puis le champ « Le type du résultat est : » et sur la ligne suivante : « [true] », « [false] » ou « [oclUndefined] ».
Dans le cas où les résultats des évaluations des invariants est vrai dans NEPTUNE et TOPCASED sont corrects nous considérons que le test est un succès.
Dans le cas où le résultat n'est pas celui attendu et que l'outil utilisé est NEPTUNE, il m'est demandé d'éditer une fiche d'anomalie notifiant l'erreur, son emplacement dans le programme, le résultat obtenu et celui attendu.

Avant de traiter les résultats revenons sur l'exécution de TOPCASED en mode batch. L'instruction à lancer en mode ligne de commande sous Windows est la suivante :

<ECLIPSE_HOME>\eclipsec –nosplash
-application org.topcased.ocl.batch.checkfile
-model <MODEL_PATH>
-rule <OCL_RULE_FILE_PATH_1> <OCL_RULE_FILE_PATH_2>...
-output <OUTPUT_DIRECTORY_PATH>
-logmodel <OCLLOG_MODEL_NAME> *(optional)*
-report <TEMPLATE_SYMBOLIC_NAME> *(optional)*

Après plusieurs recherches sur internet nous avons lancé la commande:

> eclipsec -nosplash -data "E:\Ballarin\Mes documents\Téléchargements" -application org.topcased.ocl.batch.checkfile -model "E:\Ballarin\Mes documents\runtime-EclipseApplication\ProjetPersonne\My.personnes" -rule "E:\Ballarin\Mes documents\runtime-EclipseApplication\ProjetPersonne\ocl_rule1.ocl" -output "E:\Ballarin\Mes documents\runtime-EclipseApplication\ProjetPersonne\results" -report HTML

Avec cette commande, le système commence l'évaluation OCL mais bloque ensuite ne reconnaissant pas **'EModelElement'** du metamodèle ecore recensé dans les métamodèles supportés par TOPCASED. Cette commande ne permet pas de parser le metamodèle My.ecore. Il semble que TOPCASED n'arrive pas à accéder aux métamodèles et en particulier à celui considéré.

Après de multiples recherches infructueuses et des nombreux essais restés vains, j'ai décidé de m'adresser à M. Gabel de la société CS qui connait et maintient l'interprète OCL de

l'environnement TOPCASED (voir en annexe sa réponse) qui nous a répondu **qu'il fallait faire référence au modèle ecore stocké sur Internet et non pas dans un directory** :
L'uri de notre méta modèle était la suivante :
nsURI="**platform:/resource/mmPersonne/model/My.ecore**"
où « platform:/resource » signifiait le workspace de TOPCASED

Nous sommes donc reparti à zéro en créant un nouveau projet « tutu » avec un nouveau modèle ecore « Cec.ecore » faisant référence à nsURI=**http://Cec.ecore/1.0**
Suivant ses conseils nous avons régénéré le générateur de modèle ainsi que les plug-ins java correspondants. Nous avons relancé la commande et nous nous sommes aperçu que le résultat n'était pas plus probant. A ce stade, j'en ai déduit qu'il y avait des manipulations supplémentaires à faire dans TOPCASED que je ne maîtrisais pas et qui n'étaient ni dans la documentation de TOPCASED ni disponible sur Internet. Il me fallait donc rechercher de nouvelles aides mais maintenant au sein de l'IRIT pour des raisons de proximité. Cette aide m'a été apportée par Jean-Paul Bodeveix car il avait déjà été confronté au même problème. Il en a déduit que dans la ligne de commande il n'y avait rien qui faisait référence aux programme java permettant de manipuler les métamodèles que j'avais créés. Ce code, contenu dans les plugins générés automatiquement, permettait de référencer le métamodèle à l'adresse précisée lors de la création du métamodèle, dans notre cas **http://Cec.ecore/1.0** Il manquait donc une manipulation, celle d'exporter les plugins du projet dans un répertoire à part pour y faire référence lors de l'exécution de la ligne de commande,
Pour ce faire, il faut utiliser la commande « export » disponible dans l'environnement TOPCASED, puis « Plug-in Development » et enfin « Deployable plug-ins and fragments ». Cette commande permet d'exporter le plug-in dans un répertoire indépendant du workspace ou dans d'autres répertoires déjà utilisés. Cette précision n'apparait pas dans la partie aide de TOPCASED.

Cette commande a pour objet de créer une archive au format jar directement exécutable, mais fallait-il encore référencer cette archive dans la ligne de commande. Ce que nous avons fait en lançant la commande :

"c:\program files\Topcased-5.1.0\eclipsec" -Xmx1024m -clean
-Dorg.eclipse.equinox.p2.reconciler.dropins.directory="E:\Ballarin\Mes documents\plugins" -nosplash -data "E:\Ballarin\Mes documents\Téléchargements"

-application org.topcased.ocl.batch.checkfile -model "E:\Ballarin\Mes documents\runtime-EclipseApplication\tutupersonne\oclsrc\Cec.gens" -rule "E:\Ballarin\Mes documents\runtime-EclipseApplication\tutupersonne\oclsrc\ocl_rule1.ocl" -output "E:\Ballarin\Mes documents\runtime-EclipseApplication\tutupersonne\results" -logmodel resultat.txt

Ou la commande

-Dorg.eclipse.equinox.p2.reconciler.dropins.directory="E:\Ballarin\Mes documents\plugins"

Permet de spécifier l'endroit où se trouve l'archive. Malheureusement, nous avons lancé cette commande et le résultat n'était toujours pas probant. J.-P. **Bodeveix en a déduit qu'il fallait mettre les fichiers archives dans le répertoire des plugins de TOPCASED (C:\Program Files\Topcased-5.1.0\plugins), et là l'exécution de la requête OCL s'est bien terminée.**

L'exécution de cette commande produit deux fichiers importants :

- le fichier résultat ici resultat.txt qui est un fichier XMI contenant les résultats de l'exécution en batch et
- le journal « .log » ici dans « E:\Ballarin\Mes documents\Téléchargements\.metadata » où figure le bilan du déroulement de l'exécution de la requête OCL en batch et qui notifie entre-autre les erreurs de syntaxe des expressions OCL.

Maintenant que nous savons exécuter une requête OCL en mode batch, il nous reste à faire le lien avec Salome-TMF c'est-à-dire à faire en sorte que Salome-TMF exécute automatiquement TOPCASED et NEPTUNE pour exécuter les tests. Dans Salome_TMF nous avons observé que quand nous créons une campagne de test il faut lier Salome_TMF a un programme exécutable d'extension « .scriptengine » qui lance TOPCASED ou NEPTUNE en batch, afin d'obtenir le fichier XMI des résultats. A partir de ce fichier il faudra traiter les résultats et de les organiser.

Le mode d'emploi de Salome Tmf décrit la procédure à suivre pour automatiser les tests et utiliser Salome TMF.

Pour cela j'ai découpé les tests de M.Gogolla en 7 campagnes de test appliquées sur les deux environnements de test NEPTUNE et TOPCASED (ce qui correspond à peu près au traitement de 75 % du total des requêtes contenues dans [GKB08a]).

Pour commencer j'ai transformé le fichier PDF contenant toutes les requêtes [GKB08a] en fichier .TXT grâce au logiciel « Convert Doc » que j'ai téléchargé, ensuite j'ai modifié ce fichier pour ne garder que les requêtes OCL.

J'ai effectué quelques petites modifications propres à la syntaxe de chacun des environnements.
Pour Topcased par exemple : j'ai remplacé « Person » par « gens ::Personne » dans le contexte, c'est-à-dire « context » suivi de Nom du package ::nom de la classe

De même dans Neptune : j'ai remplacé « Person » par « mmCec ::Cec ::Personne » soit dans le contexte le nom du métamodèle suivi du nom du modèle et suivi du nom du paquetage.

Dans les deux environnements :
J'ai remplacé « self.civstat » par « self.civstat=Civstatus :: » car dans mon métamodèle j'ai créé une énumération pour décrire le statut civil
Ensuite j'ai découpé le fichier ainsi obtenu et traitant de chaque aspect à tester (Bx) en sept fichiers appelés campagnes de tests.
J'ai testé ces fichiers interactivement sur les outils pour corriger les erreurs de syntaxe et pouvoir les compiler correctement. (Pour TOPCASED les requêtes contenant des « iterate » avec deux itérateurs (qui ne compilent pas comme vu précédemment) ont été mises en commentaire pour pouvoir les tester ensuite sur NEPTUNE. Pour la campagne B1 j'ai créé un fichier « plantagecompil_topcased.txt » (voir en annexe) contenant les requêtes qui ne passent pas à la compilation avec pour chaque requête le message d'erreur de compilation).
Ensuite j'ai créé dans Salome TMF les différents environnements de test chacun rattaché à un programme Java et un programme exécutable (rattaché à une campagne) permettant d'évaluer en batch chaque fichier Ocl (grâce au paramètre TEST_DRIVER_NAME)
J'ai ensuite créé les familles de tests contenant des suites et des tests rattachés chacun à script d'initialisation de l'outil TOPCASED et NEPTUNE (en Beanshell).
J'ai créé des campagnes de test, importé les tests depuis les familles et lancé les exécutions (une par campagne (Bx) et par outil (TOPCASED et NEPTUNE).
A chaque résultat d'exécution j'ai rattaché le fichier des résultats des requêtes, le modèle utilisé et le fichier des requêtes que j'ai soumises aux deux outils.
Le fichier des résultats dans TOPCASED étant un fichier .txt mais ayant le contenu d'un fichier XMI, il suffit pour obtenir une présentation conviviale des résultats de renommer ce fichier .Txt en .Xmi, ensuite d'ouvrir Excel et d'ouvrir ce fichier Xmi dans Excel. On obtient un fichier Excel avec des entêtes de colonnes avec notamment le nom de l'invariant et le résultat, ce qui permet pour NEPTUNE et TOPCASED de faire des croisements de fichiers pour analyser les résultats.

En ce qui concerne NEPTUNE, le fichier résultat a comme extension .cvs, on peut donc l'ouvrir avec Excel et reformater la présentation avec « Données » puis « Convertir » pour séparer les différents champs en colonnes. (Voir Annexe Comment formater le fichier Neptune)

Copier ensuite cette feuille de calcul dans une feuille d'un nouveau fichier excel en lui donnant le nom « Résultats Neptune ». Ouvrir ensuite le fichier résulat topcased d'extension « xmi » par exemple respersonneocl_campagneb5bis.xmi avec excel , on obtient l'écran suivant :

Fig. 13. Boite de dialogue du fichier XMI ouvert avec Excel

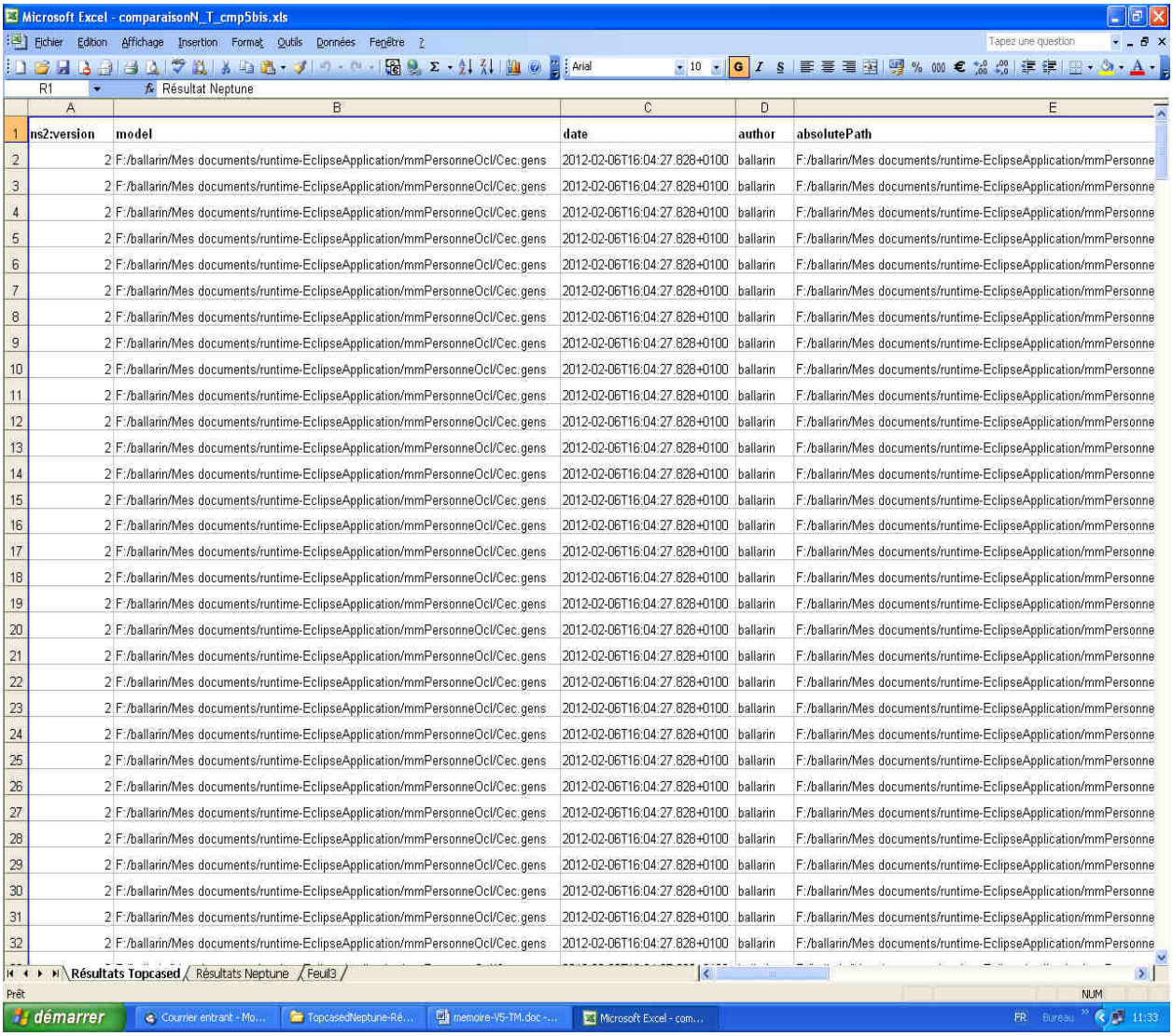
Cliquer sur ok.

Puis copier cette feuille dans le fichier nouvellement créé en le nommant « Résultats Topcased »

On obtient un fichier de la forme que l'on nome « comparaisonN_T_cmp5bis.xls » par exemple formé de la façon suivante :

- « comparaisonN_T » pour comparaison Neptune Topcased
- suivi du nom de la campagne ici « cmp5bis » pour campagne 5 bis

Le fichier obtenu est représenté à la figure suivante :



	A	B	C	D	E
1	ns2:version	model	date	author	absolutePath
2	2	F:/ballarin/Mes documents/runtime-EclipseApplication/mmPersonneOcl/Cec.gens	2012-02-06T16:04:27.828+0100	ballarin	F:/ballarin/Mes documents/runtime-EclipseApplication/mmPersonne
3	2	F:/ballarin/Mes documents/runtime-EclipseApplication/mmPersonneOcl/Cec.gens	2012-02-06T16:04:27.828+0100	ballarin	F:/ballarin/Mes documents/runtime-EclipseApplication/mmPersonne
4	2	F:/ballarin/Mes documents/runtime-EclipseApplication/mmPersonneOcl/Cec.gens	2012-02-06T16:04:27.828+0100	ballarin	F:/ballarin/Mes documents/runtime-EclipseApplication/mmPersonne
5	2	F:/ballarin/Mes documents/runtime-EclipseApplication/mmPersonneOcl/Cec.gens	2012-02-06T16:04:27.828+0100	ballarin	F:/ballarin/Mes documents/runtime-EclipseApplication/mmPersonne
6	2	F:/ballarin/Mes documents/runtime-EclipseApplication/mmPersonneOcl/Cec.gens	2012-02-06T16:04:27.828+0100	ballarin	F:/ballarin/Mes documents/runtime-EclipseApplication/mmPersonne
7	2	F:/ballarin/Mes documents/runtime-EclipseApplication/mmPersonneOcl/Cec.gens	2012-02-06T16:04:27.828+0100	ballarin	F:/ballarin/Mes documents/runtime-EclipseApplication/mmPersonne
8	2	F:/ballarin/Mes documents/runtime-EclipseApplication/mmPersonneOcl/Cec.gens	2012-02-06T16:04:27.828+0100	ballarin	F:/ballarin/Mes documents/runtime-EclipseApplication/mmPersonne
9	2	F:/ballarin/Mes documents/runtime-EclipseApplication/mmPersonneOcl/Cec.gens	2012-02-06T16:04:27.828+0100	ballarin	F:/ballarin/Mes documents/runtime-EclipseApplication/mmPersonne
10	2	F:/ballarin/Mes documents/runtime-EclipseApplication/mmPersonneOcl/Cec.gens	2012-02-06T16:04:27.828+0100	ballarin	F:/ballarin/Mes documents/runtime-EclipseApplication/mmPersonne
11	2	F:/ballarin/Mes documents/runtime-EclipseApplication/mmPersonneOcl/Cec.gens	2012-02-06T16:04:27.828+0100	ballarin	F:/ballarin/Mes documents/runtime-EclipseApplication/mmPersonne
12	2	F:/ballarin/Mes documents/runtime-EclipseApplication/mmPersonneOcl/Cec.gens	2012-02-06T16:04:27.828+0100	ballarin	F:/ballarin/Mes documents/runtime-EclipseApplication/mmPersonne
13	2	F:/ballarin/Mes documents/runtime-EclipseApplication/mmPersonneOcl/Cec.gens	2012-02-06T16:04:27.828+0100	ballarin	F:/ballarin/Mes documents/runtime-EclipseApplication/mmPersonne
14	2	F:/ballarin/Mes documents/runtime-EclipseApplication/mmPersonneOcl/Cec.gens	2012-02-06T16:04:27.828+0100	ballarin	F:/ballarin/Mes documents/runtime-EclipseApplication/mmPersonne
15	2	F:/ballarin/Mes documents/runtime-EclipseApplication/mmPersonneOcl/Cec.gens	2012-02-06T16:04:27.828+0100	ballarin	F:/ballarin/Mes documents/runtime-EclipseApplication/mmPersonne
16	2	F:/ballarin/Mes documents/runtime-EclipseApplication/mmPersonneOcl/Cec.gens	2012-02-06T16:04:27.828+0100	ballarin	F:/ballarin/Mes documents/runtime-EclipseApplication/mmPersonne
17	2	F:/ballarin/Mes documents/runtime-EclipseApplication/mmPersonneOcl/Cec.gens	2012-02-06T16:04:27.828+0100	ballarin	F:/ballarin/Mes documents/runtime-EclipseApplication/mmPersonne
18	2	F:/ballarin/Mes documents/runtime-EclipseApplication/mmPersonneOcl/Cec.gens	2012-02-06T16:04:27.828+0100	ballarin	F:/ballarin/Mes documents/runtime-EclipseApplication/mmPersonne
19	2	F:/ballarin/Mes documents/runtime-EclipseApplication/mmPersonneOcl/Cec.gens	2012-02-06T16:04:27.828+0100	ballarin	F:/ballarin/Mes documents/runtime-EclipseApplication/mmPersonne
20	2	F:/ballarin/Mes documents/runtime-EclipseApplication/mmPersonneOcl/Cec.gens	2012-02-06T16:04:27.828+0100	ballarin	F:/ballarin/Mes documents/runtime-EclipseApplication/mmPersonne
21	2	F:/ballarin/Mes documents/runtime-EclipseApplication/mmPersonneOcl/Cec.gens	2012-02-06T16:04:27.828+0100	ballarin	F:/ballarin/Mes documents/runtime-EclipseApplication/mmPersonne
22	2	F:/ballarin/Mes documents/runtime-EclipseApplication/mmPersonneOcl/Cec.gens	2012-02-06T16:04:27.828+0100	ballarin	F:/ballarin/Mes documents/runtime-EclipseApplication/mmPersonne
23	2	F:/ballarin/Mes documents/runtime-EclipseApplication/mmPersonneOcl/Cec.gens	2012-02-06T16:04:27.828+0100	ballarin	F:/ballarin/Mes documents/runtime-EclipseApplication/mmPersonne
24	2	F:/ballarin/Mes documents/runtime-EclipseApplication/mmPersonneOcl/Cec.gens	2012-02-06T16:04:27.828+0100	ballarin	F:/ballarin/Mes documents/runtime-EclipseApplication/mmPersonne
25	2	F:/ballarin/Mes documents/runtime-EclipseApplication/mmPersonneOcl/Cec.gens	2012-02-06T16:04:27.828+0100	ballarin	F:/ballarin/Mes documents/runtime-EclipseApplication/mmPersonne
26	2	F:/ballarin/Mes documents/runtime-EclipseApplication/mmPersonneOcl/Cec.gens	2012-02-06T16:04:27.828+0100	ballarin	F:/ballarin/Mes documents/runtime-EclipseApplication/mmPersonne
27	2	F:/ballarin/Mes documents/runtime-EclipseApplication/mmPersonneOcl/Cec.gens	2012-02-06T16:04:27.828+0100	ballarin	F:/ballarin/Mes documents/runtime-EclipseApplication/mmPersonne
28	2	F:/ballarin/Mes documents/runtime-EclipseApplication/mmPersonneOcl/Cec.gens	2012-02-06T16:04:27.828+0100	ballarin	F:/ballarin/Mes documents/runtime-EclipseApplication/mmPersonne
29	2	F:/ballarin/Mes documents/runtime-EclipseApplication/mmPersonneOcl/Cec.gens	2012-02-06T16:04:27.828+0100	ballarin	F:/ballarin/Mes documents/runtime-EclipseApplication/mmPersonne
30	2	F:/ballarin/Mes documents/runtime-EclipseApplication/mmPersonneOcl/Cec.gens	2012-02-06T16:04:27.828+0100	ballarin	F:/ballarin/Mes documents/runtime-EclipseApplication/mmPersonne
31	2	F:/ballarin/Mes documents/runtime-EclipseApplication/mmPersonneOcl/Cec.gens	2012-02-06T16:04:27.828+0100	ballarin	F:/ballarin/Mes documents/runtime-EclipseApplication/mmPersonne
32	2	F:/ballarin/Mes documents/runtime-EclipseApplication/mmPersonneOcl/Cec.gens	2012-02-06T16:04:27.828+0100	ballarin	F:/ballarin/Mes documents/runtime-EclipseApplication/mmPersonne

Fig. 14. Fichier de comparaison des résultats de Neptune et Topcased

Une recherche croisée permet ensuite de présenter les résultats de TOPCASED et NEPTUNE par invariant, de noter les disparités et de les analyser.

J'ai utilisé Excel car il permet de faire des recherches verticales et de mettre en regard les résultats de l'évaluation des invariants dans Neptune et Topcased. Il permet aussi l'utlisation de fitres qui rendent possible la sélection des données particulières, par exemple ici les invariants qui ont un résultat d'évaluation différent de « true ».

Un extrait des deux fichiers résultats ainsi que de la comparaison des résultats se trouve en annexe.

En conclusion nous pouvons résumer notre démarche en synthétisant les outils et méthodes nécessaires à la vérification d'un interprète OCL à sa norme.

Tout d'abord les outils que nous devons utiliser sont :

- L'installation de l'outil interprète d'OCL ici NEPTUNE et TOPCASED.
- L'étude du fonctionnement de l'outil avec si nécessaire la rédaction d'un mode d'emploi de l'outil.
- Installation d'un outil de test

Méthode utilisée pour vérifier la conformité d'un outil à sa norme

- La première étape est de rechercher une étude déjà réalisée à ce sujet un benchmarking concernant la syntaxe du langage et les résultats attendus. Si elle n'existe pas, il faut la concevoir.
- Cette recherche peut s'accompagner d'une étude approfondie de la norme.
- Etudier les résultats attendus en sortie du traitement (ici la requête et le résultat de son évaluation).
- Etudier les entrées des traitements : ici des fichiers de requêtes OCL et les modèles et étudier comment les adapter à chaque environnement.
- Organiser les requêtes à tester en campagnes
- Tester ces fichiers directement sur les outils pour corriger la syntaxe.
- Pour chaque outil faire un programme Java de d'exécution de la campagne en batch et par outil, ici un pour NEPTUNE et un pour TOPCASED.
- Choisir l'outil de test en fonction de la complexité des tests à effectuer, du volume d'information à traiter, de la présentation des résultats, de la convivialité de l'outil, de la possibilité ou non d'effectuer les traitements en différé.
- Tester les programmes java d'exécution des campagnes à l'intérieur des outils de test.
- Pour cela écrire un programme d'initialisation de l'outil de test (ici programme Beanshell).
- Convertir les fichiers résultats en .cvs ou xls
- Et traiter les résultats dans Excel. Les analyser.

ANALYSE DES RESULTATS.Dans le fichier Excel, sélectionner avec le filtre toutes les règles où la case « Comparaison Neptune Topcased » contient « problème » et à partir de la norme et des requêtes sous forme de « query » dans Neptune, étudier s'il s'agit d'un bug de l'outil et quel est l'outil qui comporte une erreur et renseigner la colonne « explication ».

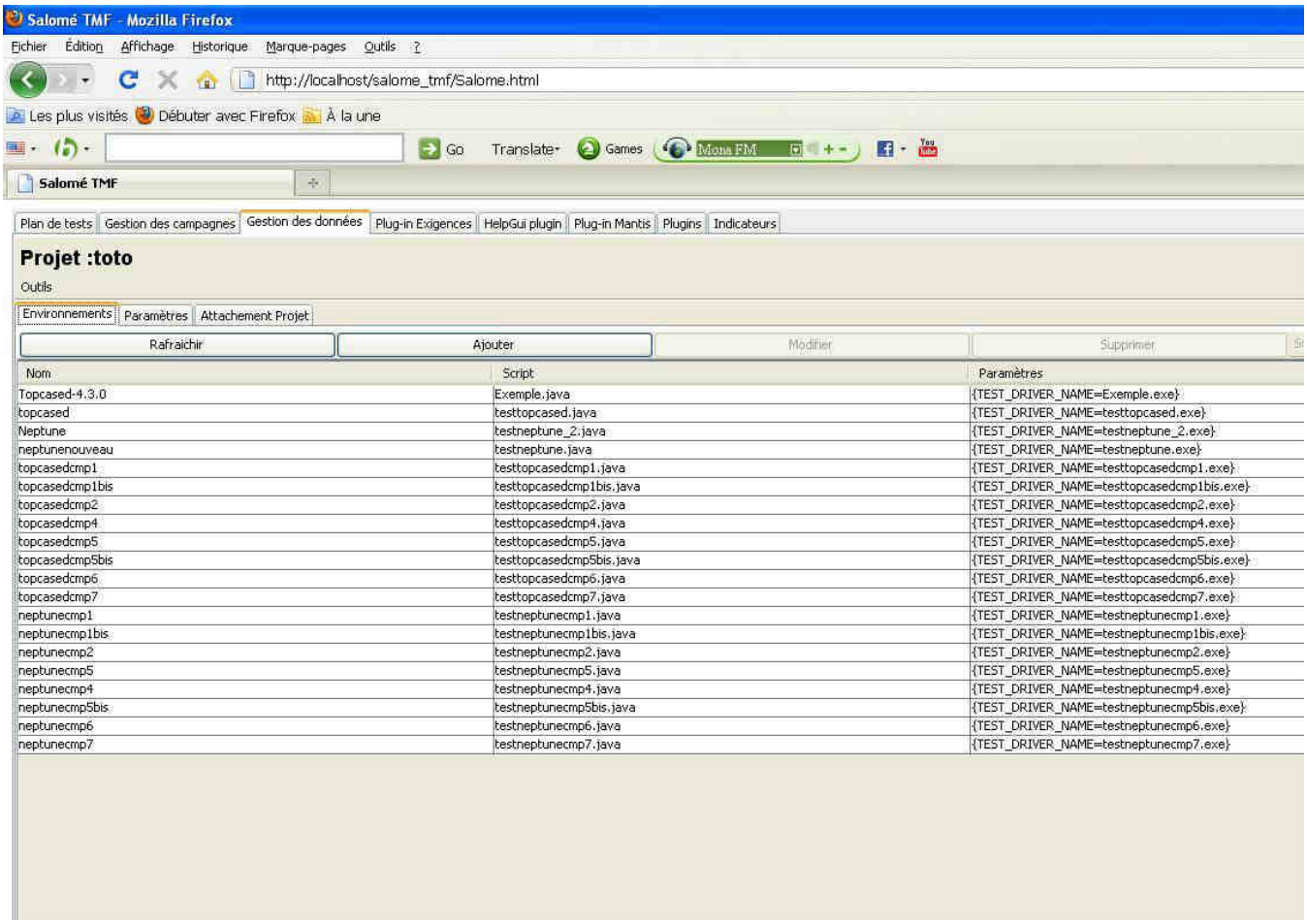
AJOUT DE NOUVELLES CAMPAGNES

Tout d'abord il est important de souligner que lancer une campagne de test avec une campagne déjà installée dans Salome TMF nécessite une demi-journée. D'autre part, il est important de rajouter de nouvelles campagnes pour les tests non couverts par M. Gogolla

Il faut **créer un fichier (Bx) appelé ocl_rule_bx.ocl** contenant les nouvelles règles à tester, tester ce fichier interactivement sur les outils pour corriger les erreurs de syntaxe et pouvoir les compiler correctement

Ensuite **il faut créer dans Salome TMF deux nouveaux environnements** de test (un pour Topcased et un pour Neptune)

Pour créer un environnement, cliquer sur l'onglet « Gestion des données » voir ci-dessous :



Nom	Script	Paramètres
Topcased-4.3.0	Exemple.java	{TEST_DRIVER_NAME=Exemple.exe}
topcased	testtopcased.java	{TEST_DRIVER_NAME=testtopcased.exe}
Neptune	testneptune_2.java	{TEST_DRIVER_NAME=testneptune_2.exe}
neptunenouveau	testneptune.java	{TEST_DRIVER_NAME=testneptune.exe}
topcasedcmp1	testtopcasedcmp1.java	{TEST_DRIVER_NAME=testtopcasedcmp1.exe}
topcasedcmp1bis	testtopcasedcmp1bis.java	{TEST_DRIVER_NAME=testtopcasedcmp1bis.exe}
topcasedcmp2	testtopcasedcmp2.java	{TEST_DRIVER_NAME=testtopcasedcmp2.exe}
topcasedcmp4	testtopcasedcmp4.java	{TEST_DRIVER_NAME=testtopcasedcmp4.exe}
topcasedcmp5	testtopcasedcmp5.java	{TEST_DRIVER_NAME=testtopcasedcmp5.exe}
topcasedcmp5bis	testtopcasedcmp5bis.java	{TEST_DRIVER_NAME=testtopcasedcmp5bis.exe}
topcasedcmp6	testtopcasedcmp6.java	{TEST_DRIVER_NAME=testtopcasedcmp6.exe}
topcasedcmp7	testtopcasedcmp7.java	{TEST_DRIVER_NAME=testtopcasedcmp7.exe}
neptunecmp1	testneptunecmp1.java	{TEST_DRIVER_NAME=testneptunecmp1.exe}
neptunecmp1bis	testneptunecmp1bis.java	{TEST_DRIVER_NAME=testneptunecmp1bis.exe}
neptunecmp2	testneptunecmp2.java	{TEST_DRIVER_NAME=testneptunecmp2.exe}
neptunecmp5	testneptunecmp5.java	{TEST_DRIVER_NAME=testneptunecmp5.exe}
neptunecmp4	testneptunecmp4.java	{TEST_DRIVER_NAME=testneptunecmp4.exe}
neptunecmp5bis	testneptunecmp5bis.java	{TEST_DRIVER_NAME=testneptunecmp5bis.exe}
neptunecmp6	testneptunecmp6.java	{TEST_DRIVER_NAME=testneptunecmp6.exe}
neptunecmp7	testneptunecmp7.java	{TEST_DRIVER_NAME=testneptunecmp7.exe}

Fig. 15. Création d'un nouvel environnement(1)

Cliquer ensuite sur le bouton « Ajouter **».**

Renseigner le nom de l'environnement, par exemple ici « topcasedcmp1 » puis cliquer sur « Chercher » et sélectionner le fichier de lancement en batch de la campagne ici

« testtopcasedcmp1.java ». Puis cliquer sur « Nouveau » pour créer le paramètre TEST_DRIVER_NAME et renseigner sa valeur avec le programme exécutable « testtopcasedcmp1.exe » Voir figures suivantes.

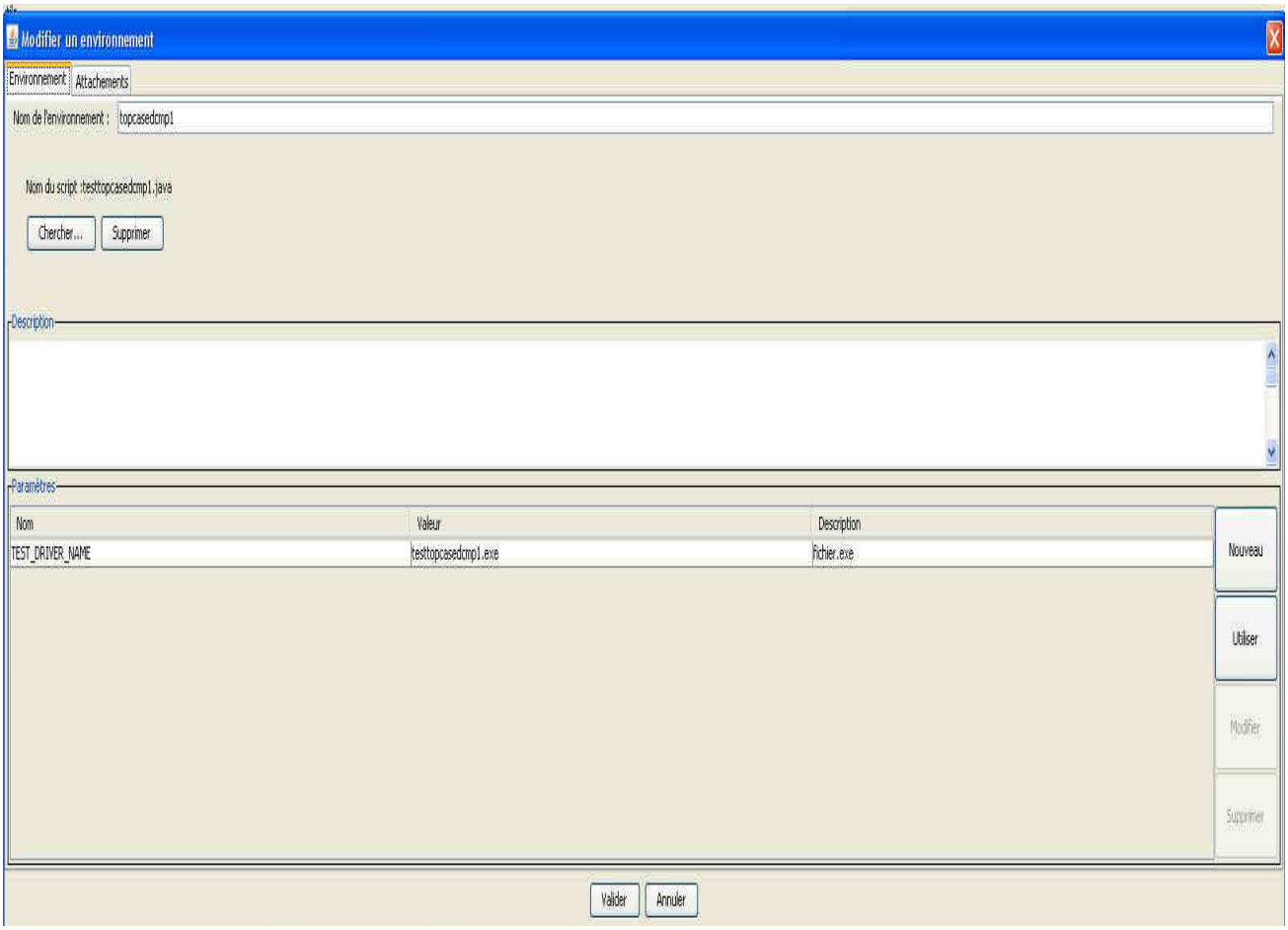


Fig. 16. Création d'un nouvel environnement (2)

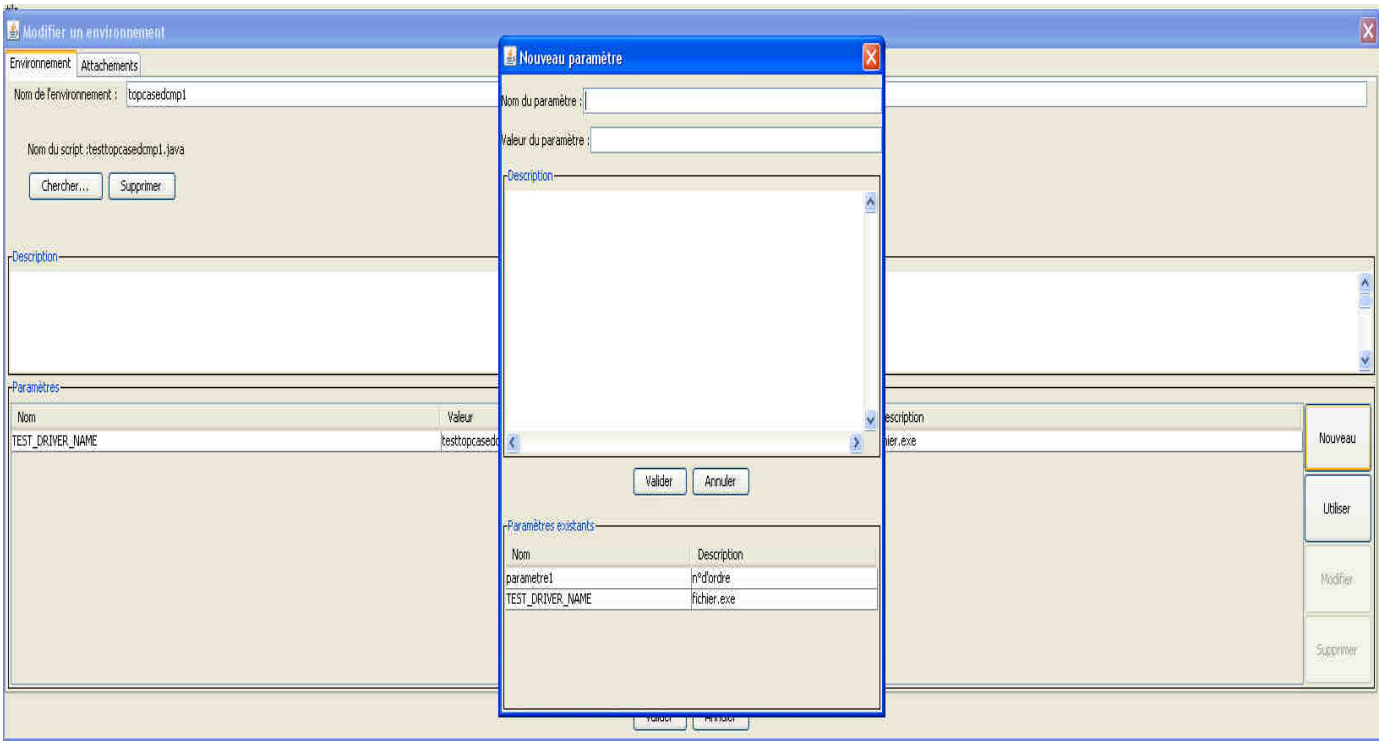


Fig. 17. Création du paramètre TEST_DRIVER_NAME(1)

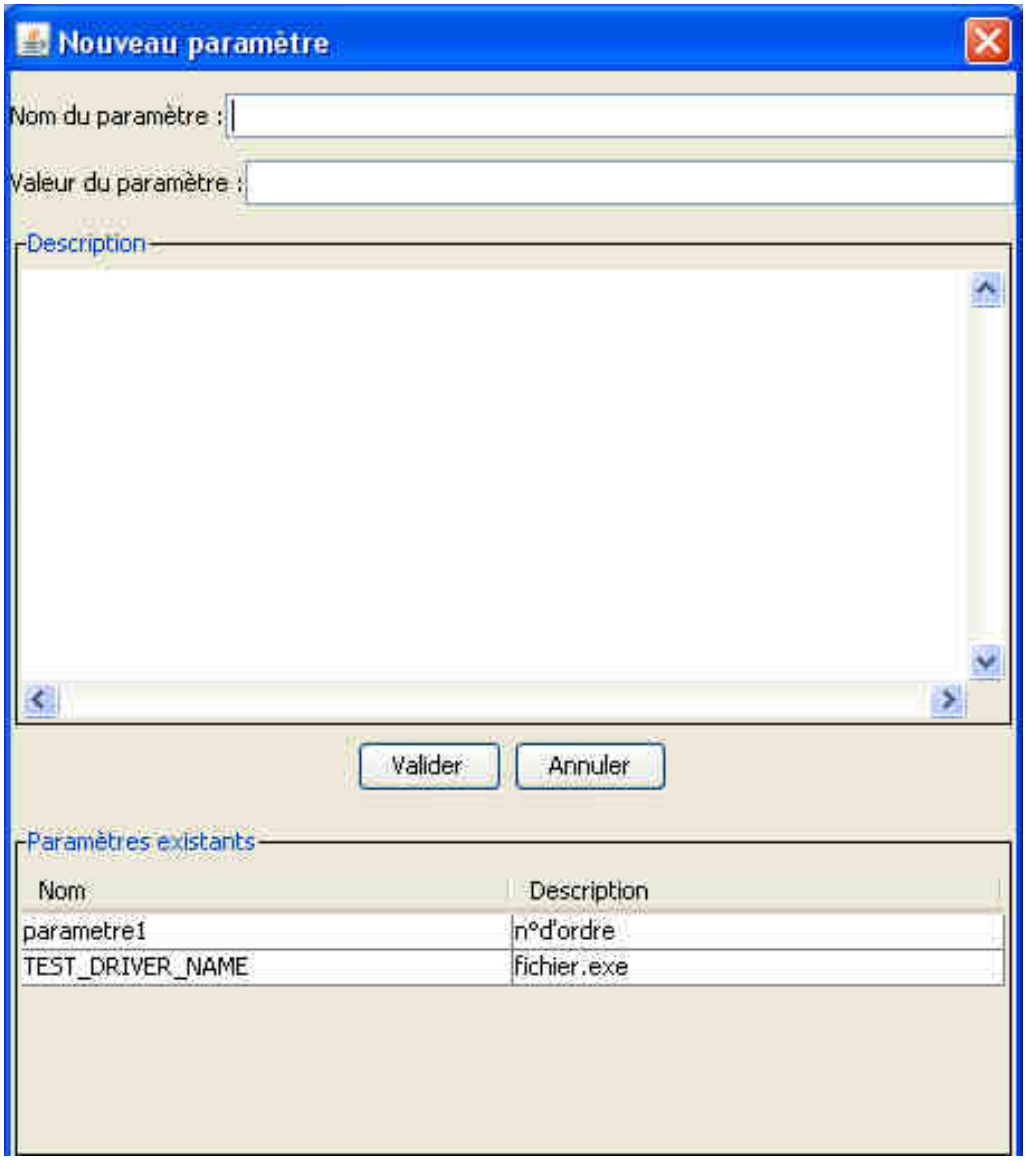


Fig. 18. Création du paramètre TEST_DRIVER_NAME(2)

A chaque environnement :

- il faut rattacher **un programme java** permettant de lancer les outils Topcased et Neptune en batch (testneptunecmp8.java et testtopcasedcmp8.java) contenant le fichier des requêtes Ocl, le modèle par exemple Cec.gens et le fichier résultat.
- un **programme exécutable** par exemple testneptunecmp8.exe et testtopcasedcmp8.exe permettant d'évaluer en batch chaque fichier Ocl (grâce au paramètre TEST_DRIVER_NAME)
- un **programme .class** par exemple testneptunecmp8.class et testtopcasedcmp8.class
- le paramètre TEST_DRIVER_NAME initialisé au nom de l'exécutable par exemple testneptunecmp8.exe ou testtopcasedcmp8.exe

Pour cela cliquer sur l'onglet « Attachements » et cliquer sur « Ajouter fichier » et sélectionner les fichiers de lancement en batch de l'environnement dont l'extension est .java, .exe, et .class.

Voir figure suivante :

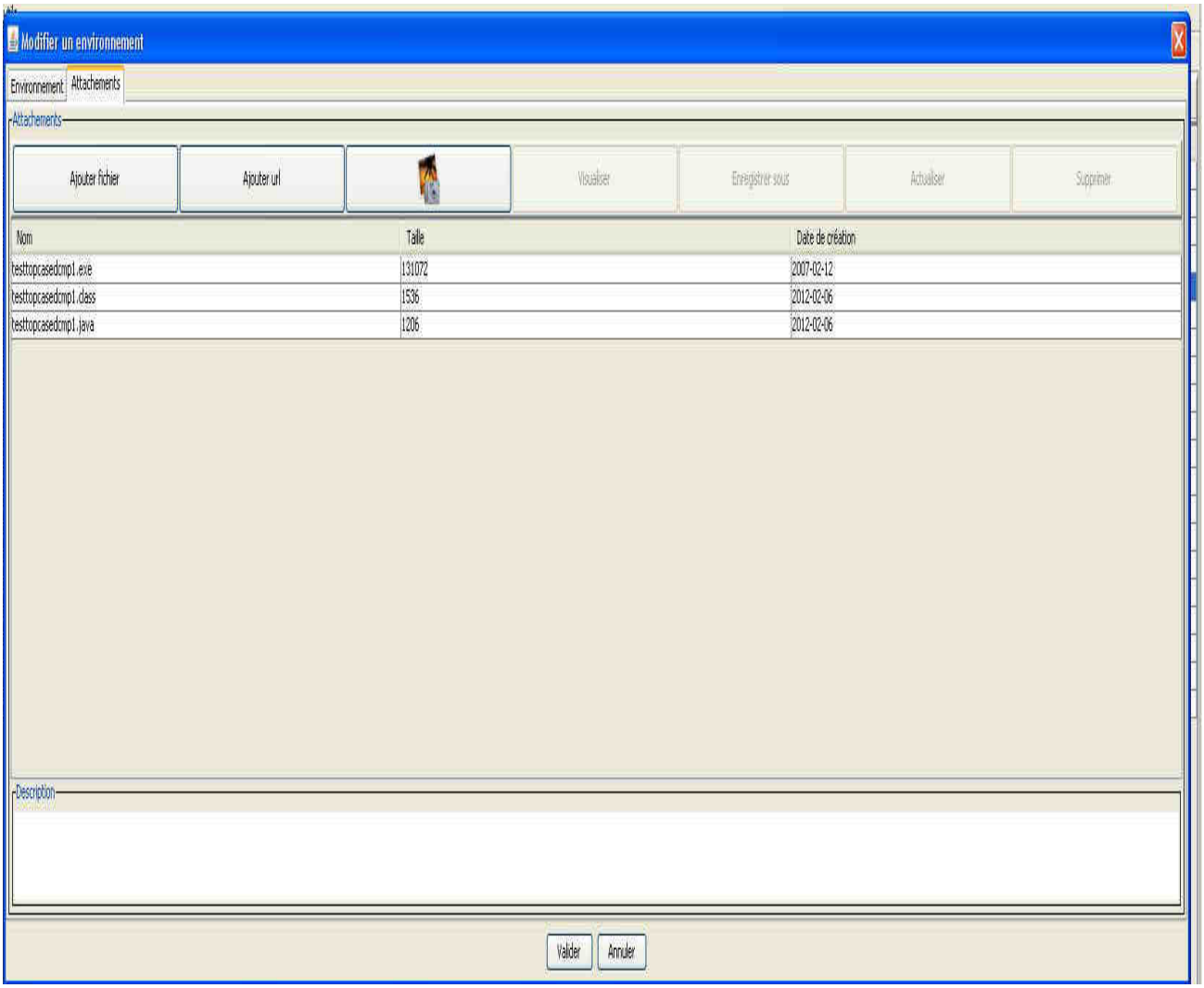


Fig. 19. Rattachement des fichiers à l'environnement

Il faut ensuite créer une famille de test rattachée à la campagne exemple famille_cmp8 contenant 2 tests d'initialisation (un pour Neptune et un pour Topcased) rattachés à deux programmes Beanshell:d'initialisation de l'exécution (testneptunecmp8.bsh et testtopcasedcmp8.bsh) contenant le paramètre TEST_DRIVER_NAME.

Pour cela cliquer sur l'onglet « Plan de tests », puis sur le bouton « Ajouter une famille »

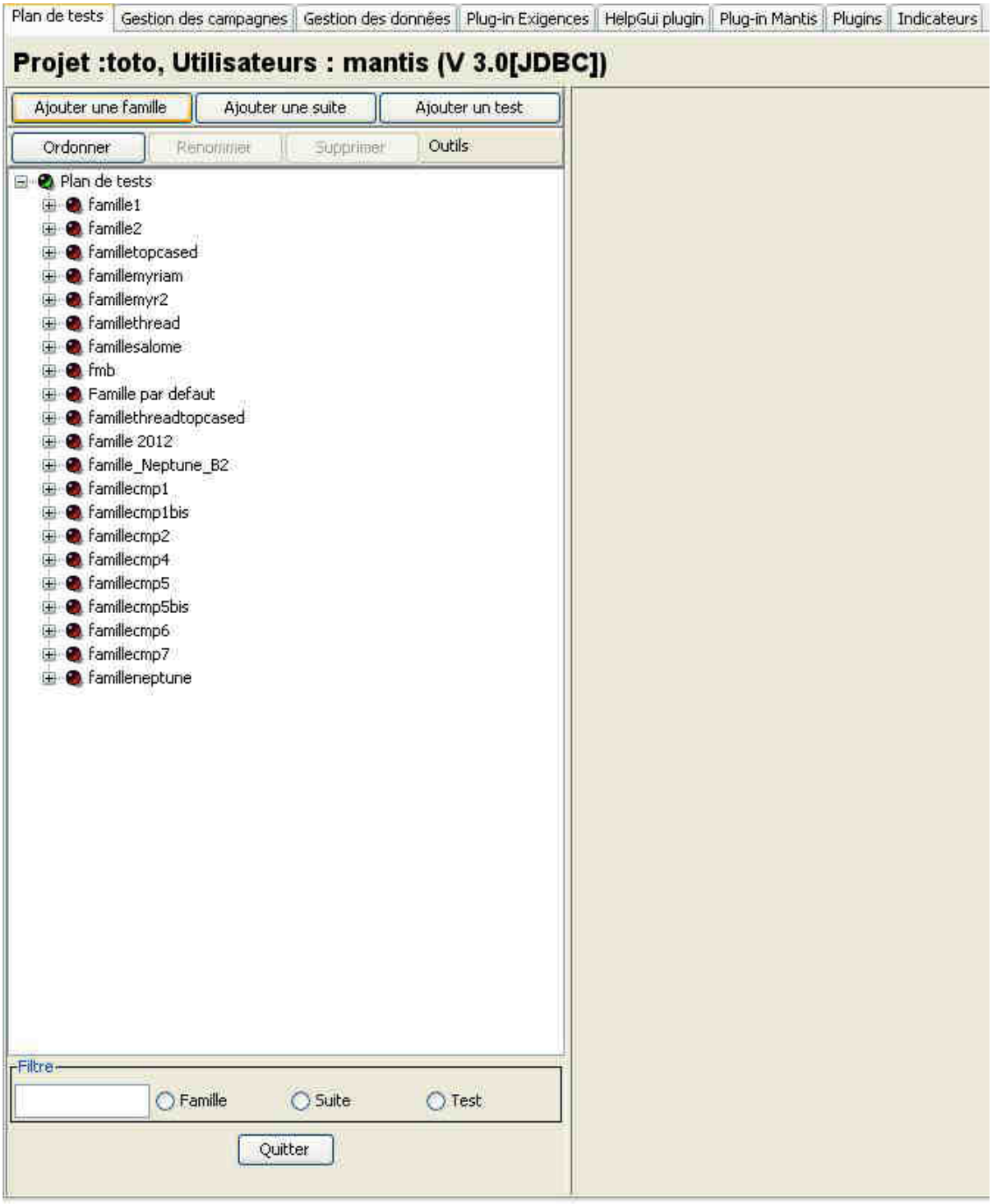


Fig. 20. Création d'une famille de tests(1)

Renseigner le nom de la famille puis sa description puis valider.

Fig. 21. Création d'une famille de tests(2)

Ajouter ensuite une suite à la famille créée.

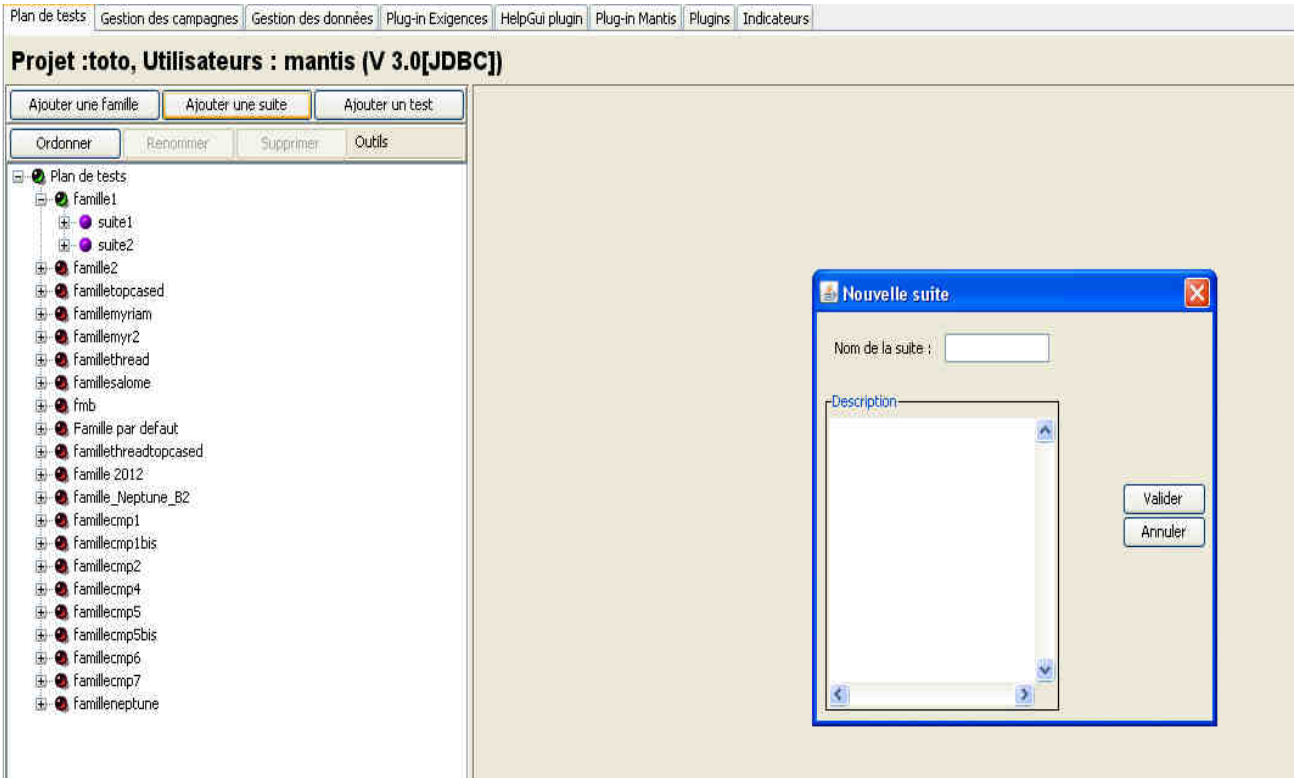


Fig. 22. Création d'une suite de tests

Puis ajouter un test Beanshell à la suite : Pour cela cliquer sur « Ajouter un test », renseigner le nom du test et choisir l'option « beanshell.TestDriver ». Voir ci-dessous

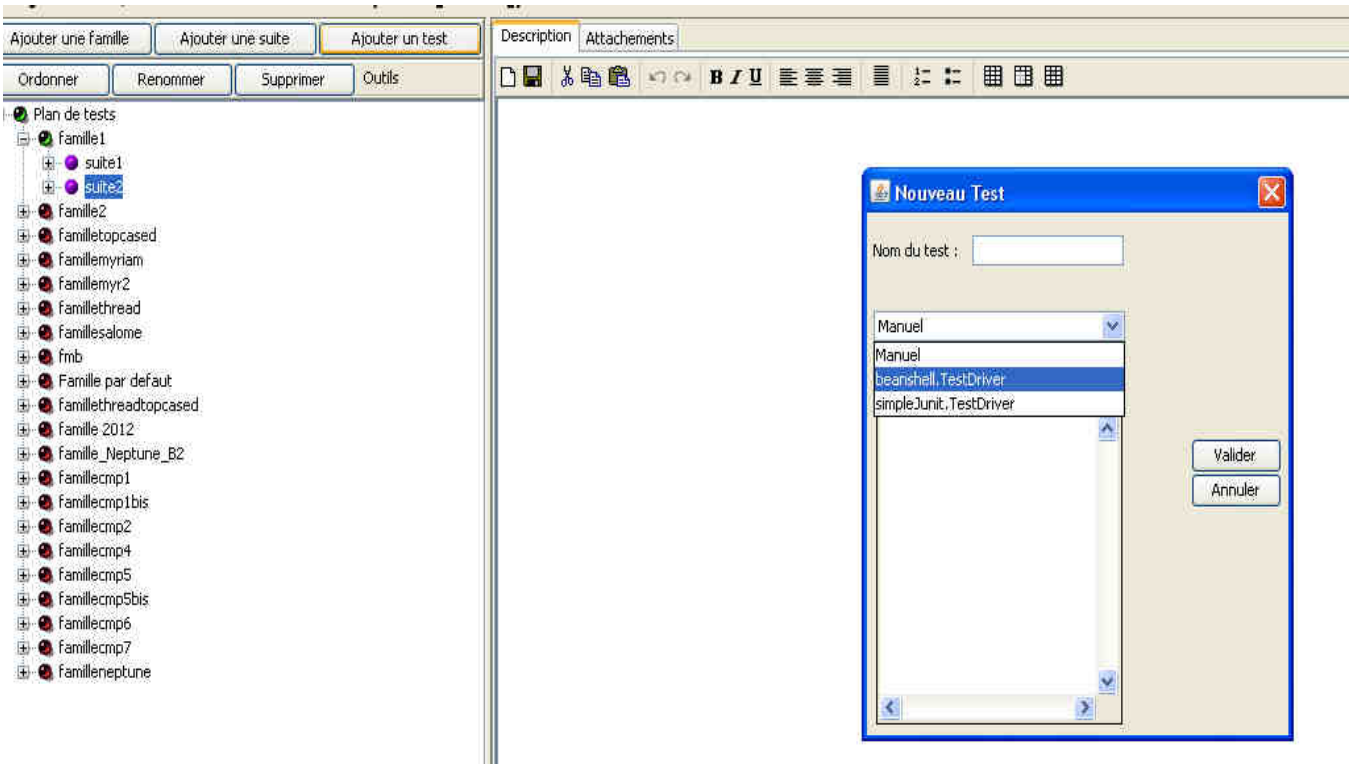


Fig. 23. Création d'un test Beanshell(1)

Rattacher le script Beanshell au test. Pour cela cliquer sur l'onglet « Script » et sur « Ajouter ». sélectionner ensuite le script Beanshell ici « testtopcased.bsh » . Voir ci-dessous.

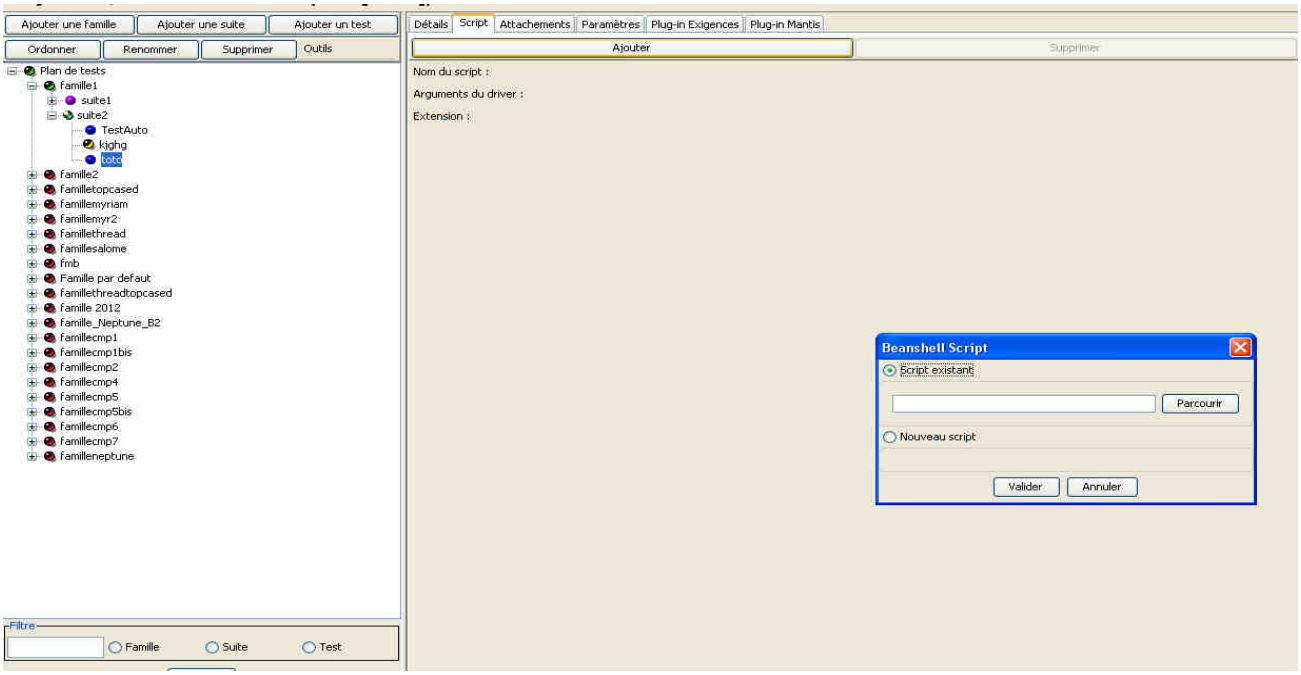


Fig. 24. Création d'un test Beanshell(2)

Rattacher le script Beanshell dans « Attachements ». Pour cela cliquer sur l'onglet « Attachements » et sur « Ajouter fichier ».

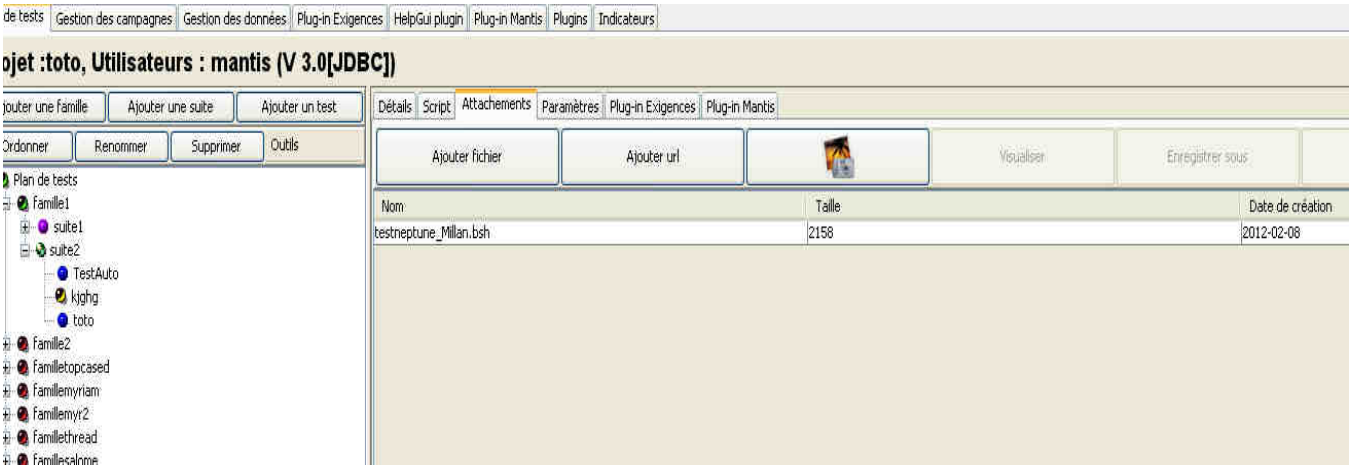


Fig. 25. Création d'un test Beanshell(3)

Il faut ensuite associer le paramètre de test (qui contient le programme exécutable de lancement en batch de l'outil) au test Beanshell. Pour cela cliquer sur « Paramètres » puis sur « Utiliser ». Voir le résultat ci-dessous.

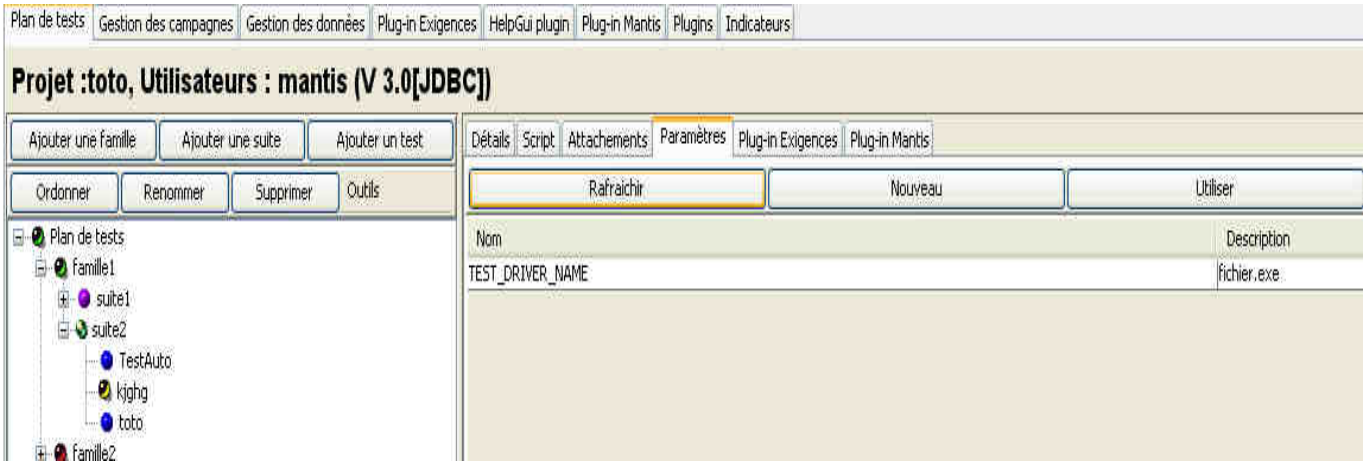


Fig. 26. Liaison du parametre TEST_DRIVER_NAME au script Beanshell

Il faut ensuite créer deux campagnes de test (une pour Neptune et une pour Topcased), importer les tests depuis les familles et lancer les exécutions

Pour cela, cliquer sur « Créer campagne » et donner un nom à la campagne.

Positonner le curseur sur la campagne et cliquer sur « importer » pour importer les tests de la famille :

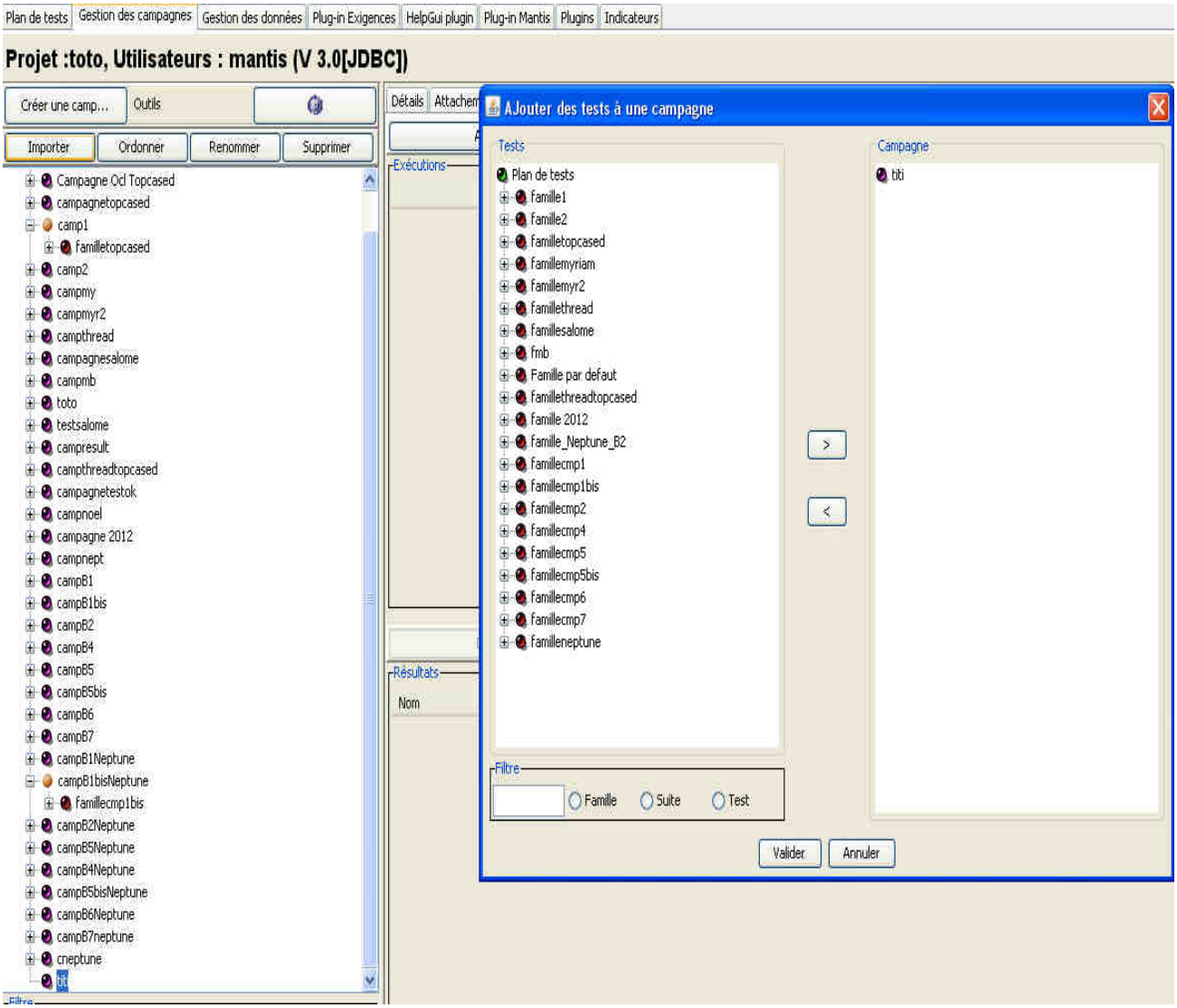


Fig. 27. Rattachement des tests à une campagne(1)

Choisir les tests à importer et valider:

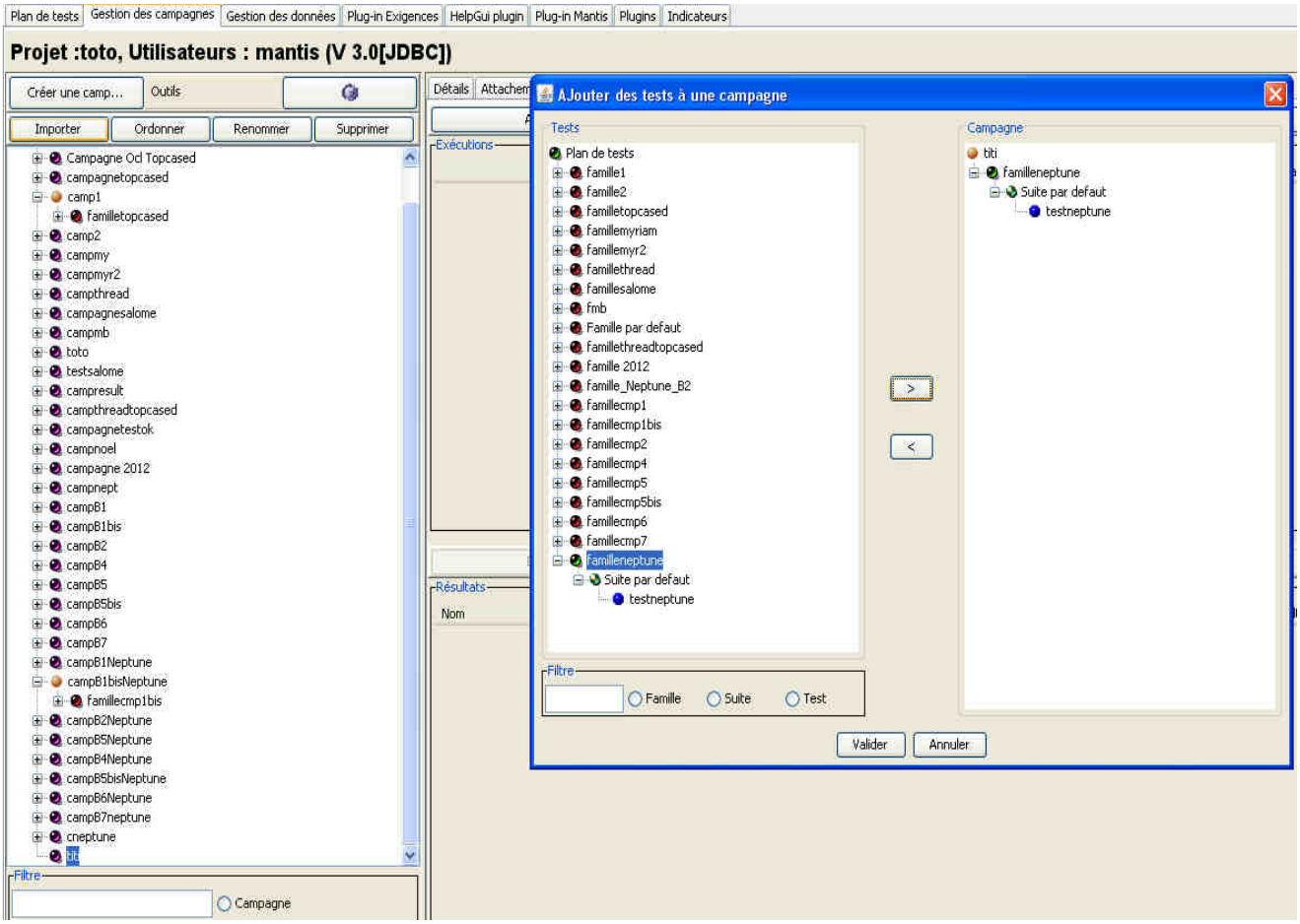


Fig. 28. Rattachement des tests à une campagne(2)

Dans une exécution il faut **rattacher l'environnement** topcasedcmp8 ou neptunecmp8 créé précédemment **et le script d'initialisation** testtopcasedcmp8.bsh ou testneptunecmp8.bsh.
Et enfin ne pas oublier de copier dans le répertoire Salome_data le fichier exécutable, le fichier. Java et le fichier .class

ne pas oublier de copier dans notre répertoire TEMP de Salome_data (F:\ballarin\TEMP\salome_data) le fichier d'extension class ici testtopcased.class

Pour cela cliquer sur l'onglet « Exécutions » Puis sur « Ajouter ». Dans le cadre « Environnement », sélectionner l'environnement (sur la figure « neptunecmp1 ») puis cliquer sur « Ajouter » dans le cadre « Scripts » et sous « Script d'initialisation » et sélectionner le script beanshell de lancement de l'environnement. Puis valider.

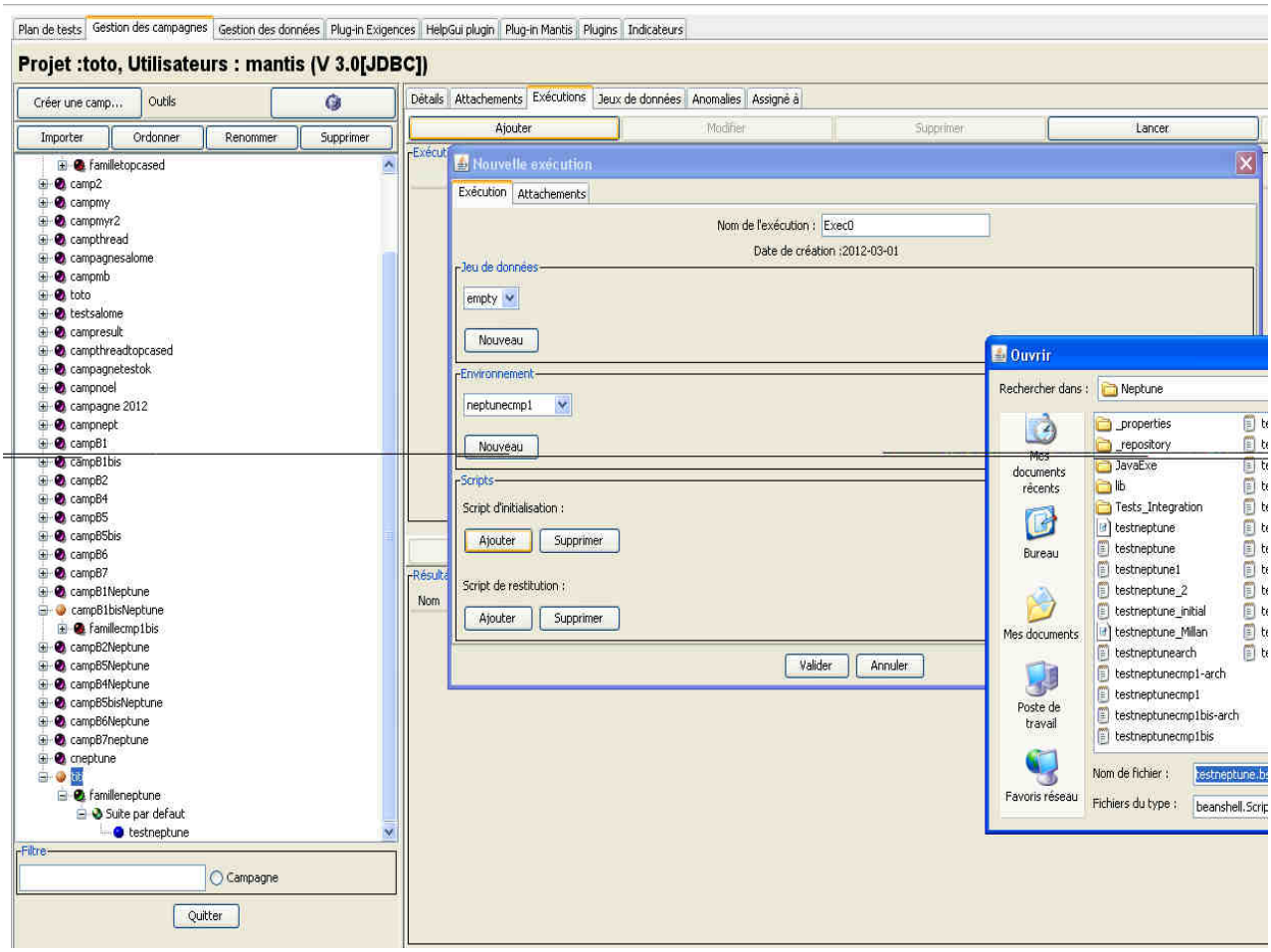


Fig. 29. Création d'une exécution

Pour lancer l'exécution cliquer sur

Fig. 30. Bouton de lancement d'une exécution

Puis sélectionner l'exécution, et saisir l'heure d'exécution puis valider.

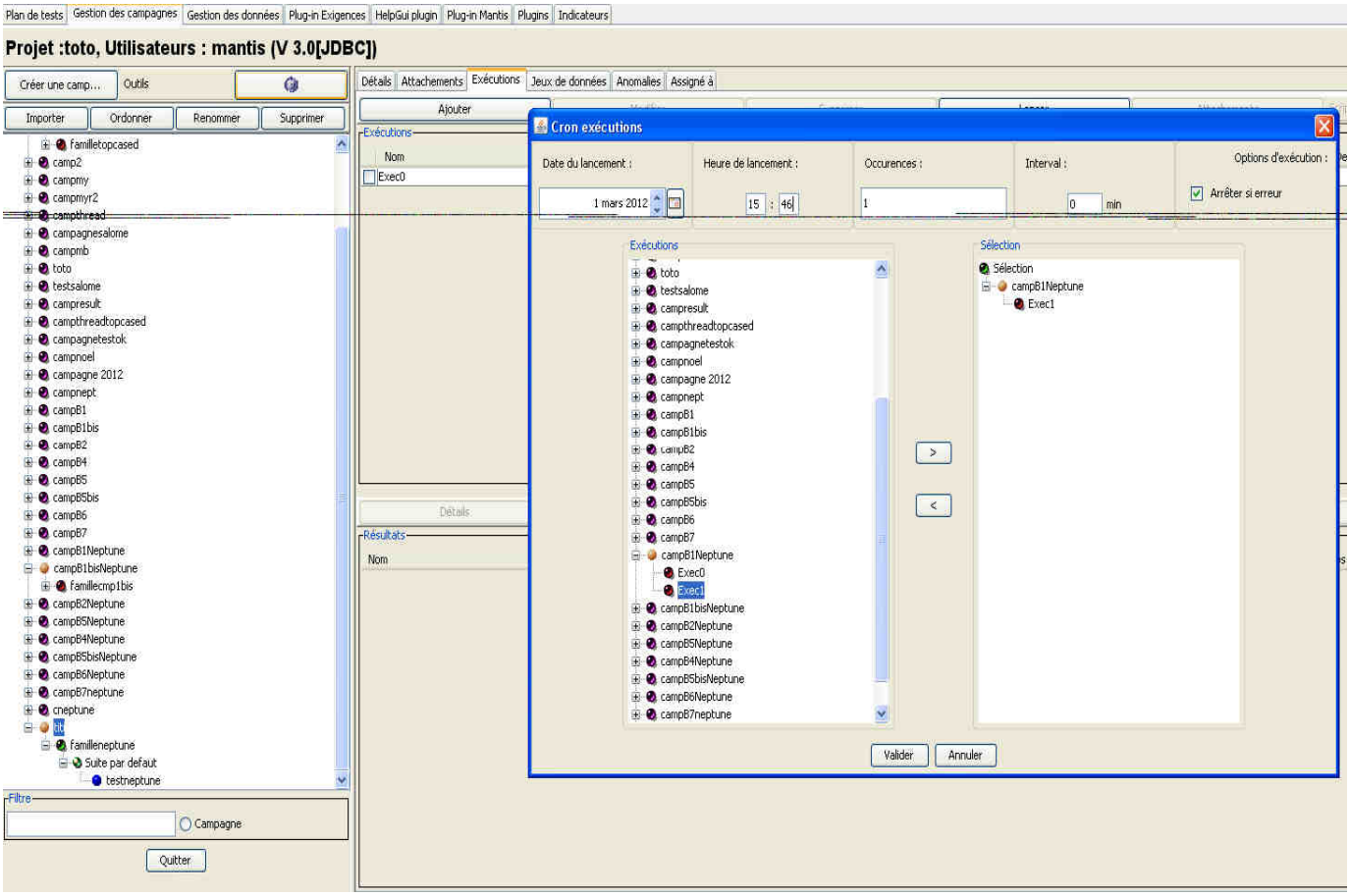


Fig. 31. Paramétrage de lancement d'une exécution

A chaque résultat d'exécution il faut rattacher le fichier des résultats des requêtes, le modèle utilisé et le fichier des requêtes que j'ai soumises aux deux outils.

Pour visualiser le résultat d'une exécution, cliquer sur la campagne et sur l'exécution ici « Exec1 », voir ci-dessous :

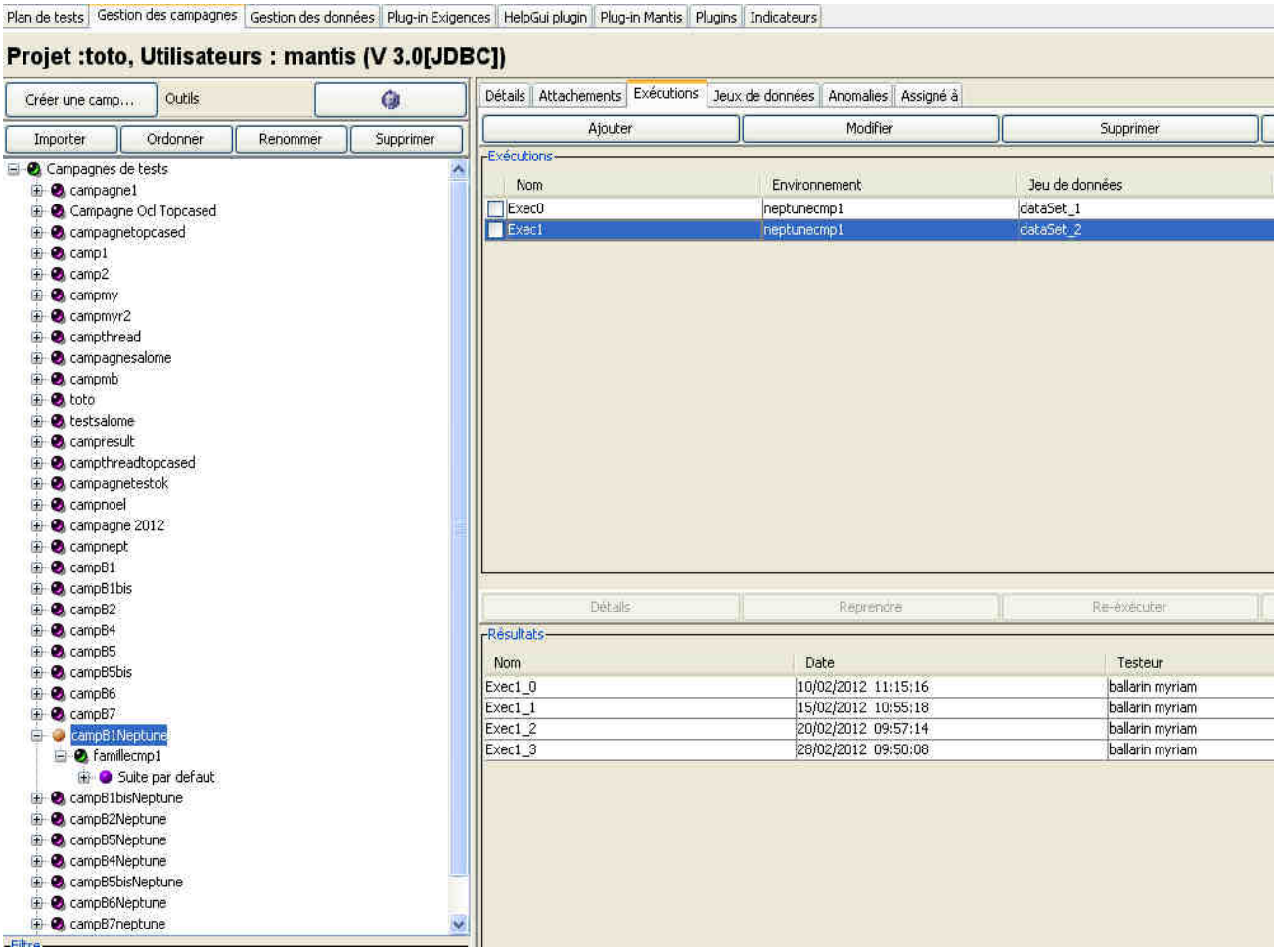


Fig. 32. Visualisation du résultat d'une exécution(1)

Puis cliquer sur le résultat de l'exécution, ici « Exec1_3 », voir ci-dessous :

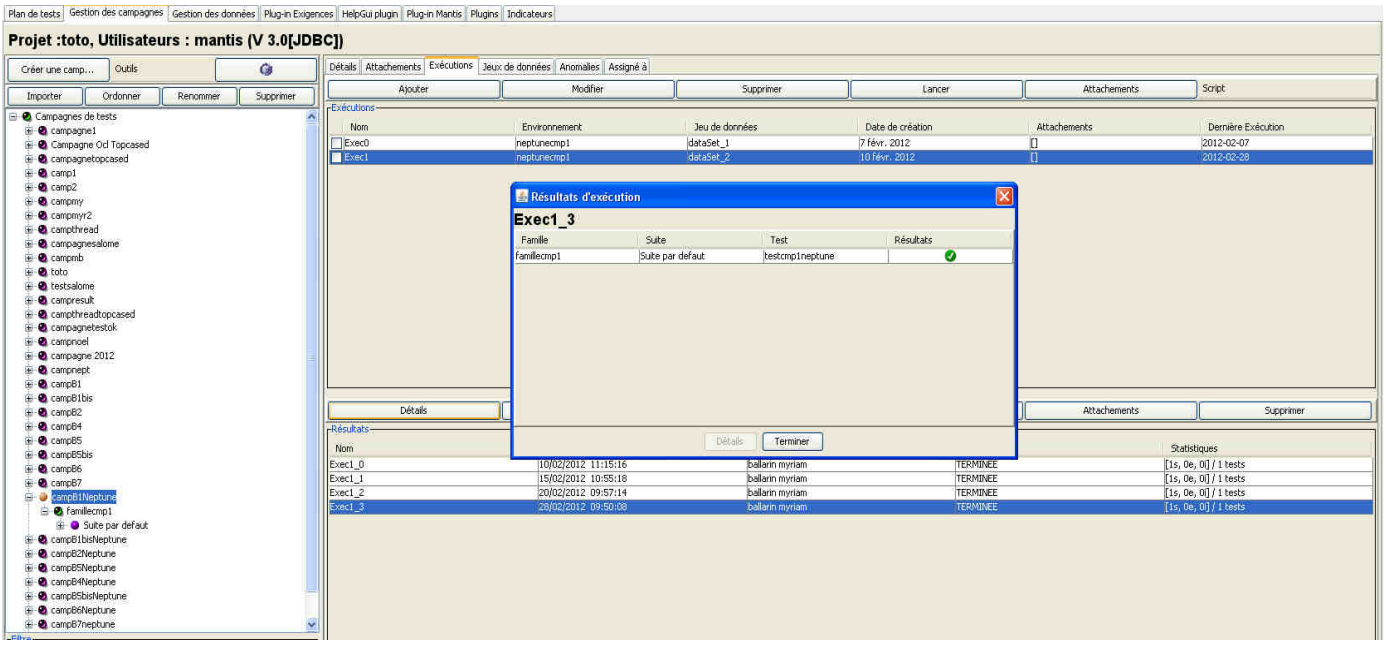


Fig. 33. Visualisation du résultat d'une exécution(2)

Pour visualiser le fichier résultat, cliquer sur « famillecmp1 » et sur « Détails »

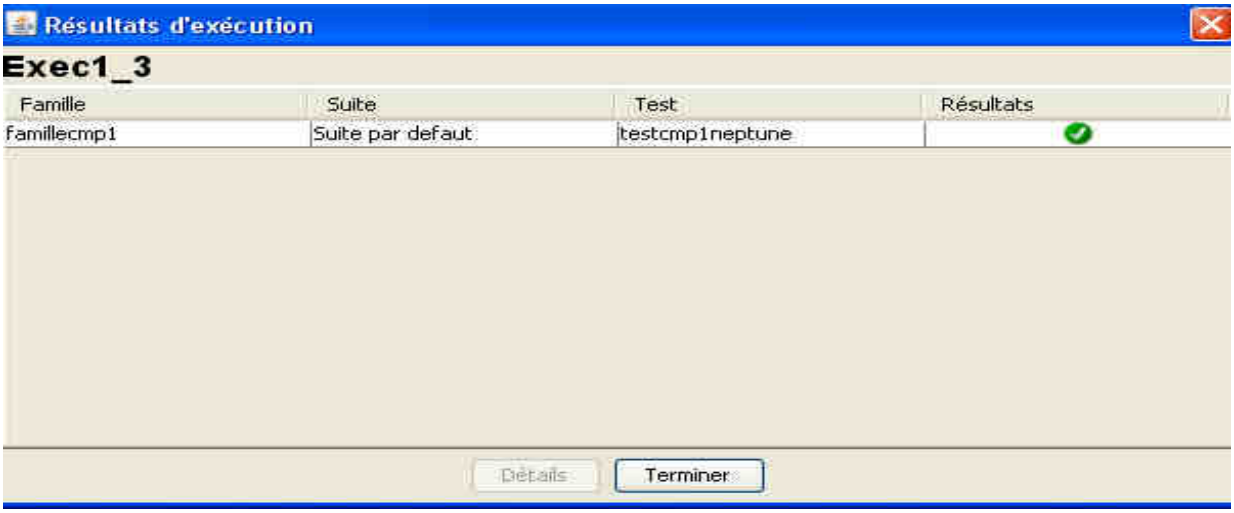


Fig. 34. Visualisation du résultat d'une exécution(3)

Le cadre suivant aparaît ensuite :

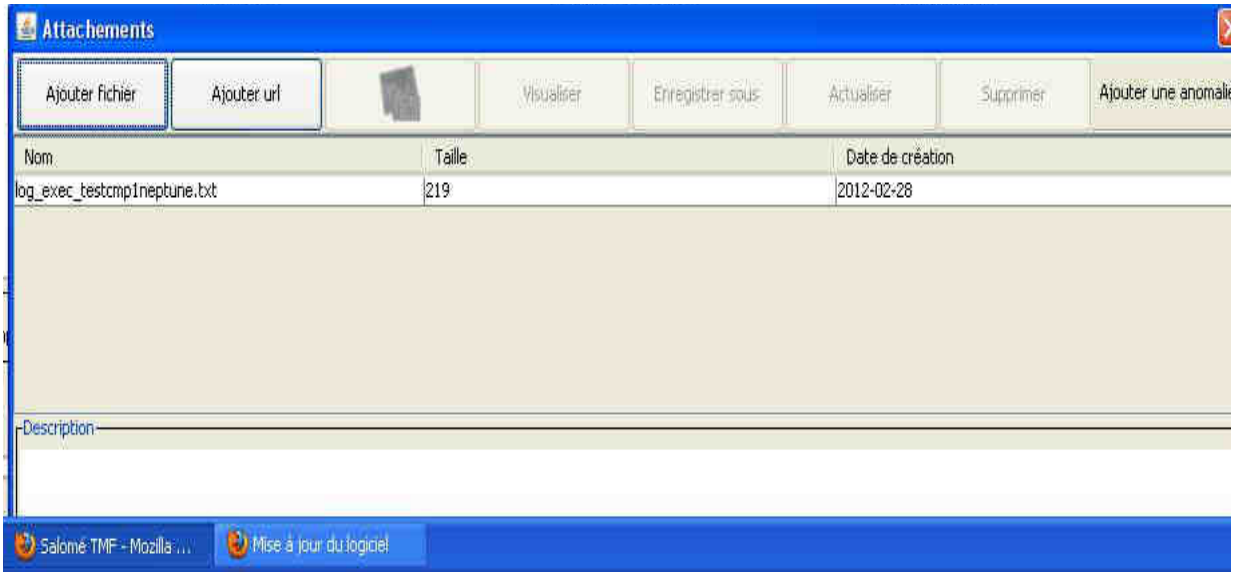


Fig. 35. Visualisation du résultat d'une exécution(4) et rattachement de fichiers à l'exécution

Cliquer sur « log_exec_testcmp1neptune.txt » et sur « Visualiser » pour visualiser le fichier résultat et sur « Ajouter fichier » pour rattacher des fichiers à l'exécution.

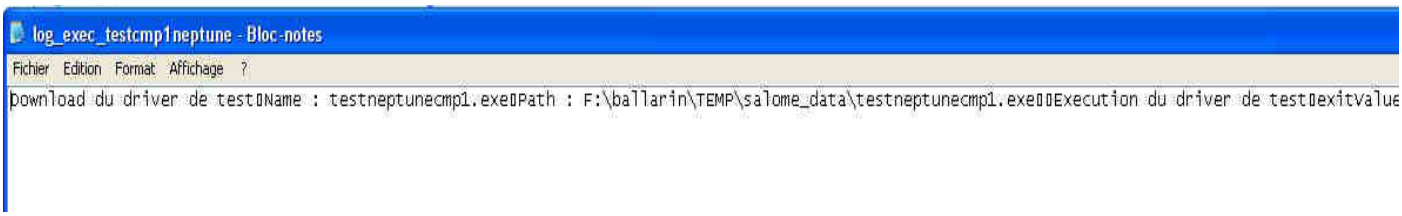


Fig. 36. Détail du fichier résultat

Outre de rajouter une nouvelle campagne on peut tester les requêtes avec d'autres outils.

AJOUT DE NOUVEAUX OUTILS

Pour rajouter un nouvel outil comme DRESDEN OCL, il faut créer un programme java qui permet de le lancer en batch, créer un nouveau programme exécutable, créer un nouveau script Beanshell et ensuite créer une nouvelle campagne de tests comme énoncé précédemment (ajout d'un nouvel environnement, d'une nouvelle famille et d'une nouvelle campagne) puis exécution de la campagne.

Maintenant que nous avons vu comment on pouvait élargir notre étude on peut s'intéresser aux résultats que nous avons actuellement.

RESULTATS DES TESTS

Durant ma campagne de tests, les résultats obtenus montrent un problème avec les valeurs « null » notamment ceux pour lesquels il y a une différence de résultat entre TOPCASED NEPTUNE et les résultats de M.Gogolla qui sont dus en majorité liés à l'évaluation des valeurs nulles.
L'invariant suivant a comme résultat « true » dans Topcased et dans [GKB08a] et « false » dans Neptune :

```
inv r288 :let c:Set(Integer) = Set{null} in
c->one(i|i<4) = (c->reject(i|i<4)->size() = c->size()-1)
```

Laquelle des réponses est juste ?
Dans la norme [OMG09] page 164 l'opération reject est définie comme suit :
reject représente le sous-ensemble du Set source pour lequel body est faux

```
source->reject(iterator | body) = source->select(iterator | not body)
```

reject peut avoir au maximum une variable iterator

Si on considère maintenant, la définition de null page 138 paragraphe OclVoid :
N'importe quelle propriété appliqué à null a comme résultat OclInvalid, excepté pour les operations oclIsUndefined() et oclIsInvalid().

```
Set {null}->reject(i : Integer | i.<(4))->size()
```

L'évaluation de null.<(4) retourne oclInvalid c'est-à-dire une valeur différente de false donc le résultat de

`Set {null}->reject(i : Integer | i.<(4))` est `Set{}` donc

`Set {null}->reject(i : Integer | i.<(4))->size() = 0.` Cela est donc égal à `Set {null}->size().-(1))`

Donc l'évaluation est correcte.

Prenons maintenant

`let c : Set(Integer) = Set {null} in c->one(i : Integer | i.<(4))`

La définition de one est (voir page 162) :

A comme résultat true s'il y a exactement un élément dans la collection source pour lequel body est true

`source->one(iterator | body) = source->select(iterator | body)->size() = 1`

one peut avoir au maximum une variable iterator.

Comme ci-dessus, null.<(4) retourne oclInvalid et non true donc

let c : Set(Integer) = Set {null} in c->one(i : Integer | i.<(4)) retourne false.

Il en résulte que :

`let c : Set(Integer) = Set {null} in c->one(i : Integer | i.<(4)).=(c->reject(i : Integer | i.<(4))->size().=(c->size().-(1)))`

retourne false.

Ce problème est récurrent dans les bases de données qui elles aussi manipulent des valeurs nulles.

Or pour être conforme à la norme null < 4 retourne oclInvalid et non faux comme dans Topcased.

Dans Neptune nous observons que nous avons environ 20% de résultats ayant pour valeur Invalid qui correspondent à false dans Topcased.

Il semblerait le problème dans l'implantation des outils USE et TOPCASED vienne du fait que pour eux, une expression booléenne a uniquement deux valeurs vrai et faux donc il implante reject à partir du résultat du select. Cette implantation est incorrecte car en réalité une expression booléenne est constituée de trois valeurs (vrai, faux et inconnu) et je rajouterais même 4 valeurs : vrai, faux, invalide et inconnu qui correspondent respectivement à true, false, oclInvalid et null (oclUndefined) On pourrait raisonner en termes

- de logique ternaire : **la logique ternaire**, ou logique 3 états, est une branche du calcul des propositions qui étend l'algèbre de Boole, en considérant en plus des états VRAI et FAUX l'état INCONNU et non pas invalid

- ou polyvalente : les **logiques polyvalentes** (ou multivalentes, ou multivaluées) sont des alternatives à la logique classique aristotélicienne, bivalente, dans laquelle toute proposition doit être soit vraie soit fausse. Elles sont apparues à partir des années 1920, surtout à la suite des travaux du logicien polonais Jan Łukasiewicz. Elles sont principalement étudiées au niveau du seul calcul propositionnel (ou **calcul propositionnel** est une théorie logique qui définit les lois formelles du raisonnement) et peu au niveau du calcul des prédicats (expression vrai ou fausse selon la valeur attribuée aux variables qu'elle contient voir définitions dans wikipédia).

Dans la logique ternaire de Stephen Cole Kleene, les tables de vérité des fonctions de base sont les suivantes :

Tableau VII: Table de vérité de la logique Ternaire.

A	***B***	***A* OU *B***	***A* ET *B***	**NON *A***	***A* implique *B***
Vrai	Vrai	Vrai	Vrai	Faux	Vrai
Vrai	Inconnu	Vrai	Inconnu	Faux	Inconnu
Vrai	Faux	Vrai	Faux	Faux	Faux
Inconnu	Vrai	Vrai	Inconnu	Inconnu	Vrai
Inconnu	Inconnu	Inconnu	Inconnu	Inconnu	Inconnu
Inconnu	Faux	Inconnu	Faux	Inconnu	Inconnu
Faux	Vrai	Vrai	Faux	Vrai	Vrai
Faux	Inconnu	Inconnu	Faux	Vrai	Vrai
Faux	Faux	Faux	Faux	Vrai	Vrai

Mais peut-on l'adapter à OCL ? Revenons à notre cas :

Null <4 retourne oclInvalid. Si on traduit çà veut dire que oclUndefined < 4 est invalide car la norme dit " Any property call applied on null results in OclInvalid, except for the operation oclIsUndefined() and oclIsInvalid()" à la page 138 paragraphe « 11.2.3 OclVoid ».

Or dans l'absolu çà ne me semble pas vrai car quelque chose qui est inconnu peut être < 4, la réponse est vrai et fausse à la fois : elle est « INCONNU » donc le résultat de Null < 4 devrait être Null c'est-à-dire « inconnu »..

Il semblerait donc qu'il y ait un grand vide à ce niveau-là dans le traitement des bases de données, le cas du résultat invalid n'a pas été prévu.

La **logique ternaire** a trois états VRAI et FAUX et l'état INCONNU qui correspond à null. Il faudrait pouvoir remplacer « invalid » par « null ».

```
Set {null}->reject(i : Integer | i.<(4) retourne Set{}
```

L'évaluation de null.<(4) retourne oclInvalid c'est-à-dire une valeur différente de false, si on décrète qu'on remplace le résultat oclInvalid par null on le même résultat puisque c'est différent de false.

Donc en réalité le résultat dans les deux cas :

```
Set {null}->reject(i : Integer | i.<(4)
```
est set{}

Notre problème est cette phrase de la norme: " Any property call applied on null results in OclInvalid, except for the operation oclIsUndefined() and oclIsInvalid()". On a recherché dans la norme ce que signifie "property call " et inexorablement, on tombe sur les attributs, les opérations, etc. Nos résultats semblent corrects vis à vis de la norme. Mais est-ce que cela ne dénature pas le langage Ocl.

Faut-il faire comme dans TOPCASED considérer que tout résultat oclInvalid est faux, ou vouloir être conforme à la norme et rajouter le résultat oclInvalid ?

Je pense que la solution la plus avant-gardiste que l'on puisse adopter à ce jour est de considérer le cas oclInvalid avec la logique ternaire puisqu'elle est connue.

Le second problème que nous avons rencontré concerne l'indéterminisme de l'ordre d'une séquence en OCL.

La norme dit concernant les Set(t) :

"Note that the semantics of the operation asSequence is nondeterministic. Any sequence containing only the elements of the source set (in arbitrary order) satisfies the operation specification in OCL"(voir paragraphe A.2.5.6 Set Operations page 196).

M.Gogolla ainsi que les concepteurs de Topcased considèrent que Ocl est déterministe alors que ce n'est pas le cas en effet quelle relation d'ordre peut-on utiliser pour transformer un Set en Sequence, ordre croissant, décroissant, ordre alphabétique ?

Pour illustrer mon propos voici un exemple :

```
let c : Set(Integer) = Set {null, 1, -1, 3, 2, 0} in c->asSequence()->=(Sequence {c}->flatten())
```

Un ensemble est par définition non ordonné donc :

`Set {null, 1, -1, 3, 2, 0} = Set {-1, 3, 2, 0, null, 1}`

Une séquence est par définition ordonnée donc !

`Sequence {null, 1, -1, 3, 2, 0} <> Sequence {-1, 3, 2, 0, null, 1}`

Si on prend maintenant :

`c->asSequence()` on obtient `Sequence{0, 2, 3, -1, 1, null}`

si on considère maintenant :

`Sequence {c}->flatten()` on obtient `Sequence{null, 1, -1, 3, 2, 0}`

Ce qui est normal vu qu'il n'y a aucune indication sur comment se fait la transformation entre un ensemble non ordonnée et une séquence.

Je pense que la solution serait de définir et d'imposer dans la norme OCL une relation d'ordre pour les Sequence, ou de définir une opération, une syntaxe particulière pour chacune des relations d'ordre possibles.

CONCLUSION

En conclusion , on peut avancer que cette étude a permis de soulever quelques problèmes dans l'interprétation possible de la norme OCL comme le traitement des valeurs « null » et l'indéterminisme concernant les Sequence. Cette étude a aussi permis d'évaluer les outils Neptune et Topcased, de signaler certains problèmes sur Topcased, par exemple sur la syntaxe du « iterate » et le typage des données. Certes il reste encore les tests de charge que je n'ai pas pu faire par manque de temps.

Elle a aussi permis d'ouvrir une perspective : l'utilisation de la logique ternaire et même quaternaire qui permettrait d'être totalement conforme à la norme alors qu'actuellement un outil comme Topcased utilise l'algèbre de Boole c'est-à-dire la logique binaire.

En ce qui me concerne, durant ce stage d'ingénieur, j'ai mis en pratique mes connaissances théoriques acquises aux cours du Conservatoire National des Arts et Métiers, notamment en matière d'analyse UML pour la constitution des méta-modèles et des modèles, en programmation Java pour le lancement des outils Topcased et Neptune en Batch et pour la programmation Beanshell fortement inspirée de Java pour programmer l'outil salome TMF, en multiprogrammation notamment lors de la résolution d'un bug impliquant Salome TMF et Neptune.

J'ai pu transposer mes connaissances en bases de données SQL afin de les utiliser pour résoudre le traitement des requêtes OCL.

J'ai travaillé sur des outils de pointe utilisés dans l'industrie comme Topcased et Salome TMF.

J'ai mené à bien des campagnes de tests et j'ai signalé les erreurs en me référant à une norme : la norme OCL qui est une norme internationale.

Plus globalement j'ai acquis une grande autonomie : j'ai expérimenté la démarche de résoudre un problème informatique dans son intégralité, de réfléchir à une stratégie d'action, depuis l'analyse jusqu'à la programmation et les tests par exemple dans la recherche de la documentation, l'installation des outils, l'apprentissage de leur fonctionnement, la résolution des bugs, l'organisation des tests en campagnes, le traitement et notamment l'interprétation des résultats, j'ai pu amener une réflexion critique notamment sur la norme, tout cela en ayant une grande liberté d'action. Certes l'installation des outils de test notamment a été pour moi un peu difficile avec l'installation des plugins car c'est quelque chose que je n'avais jamais fait durant mon expérience professionnelle. J'ai du faire le chargement automatique de Salome Tmf uniquement pour la version 3, télécharger un nombre important de plug-in dont il fallait installer des versions qui allaient ensemble et avec Salome Tmf version 3. J'ai eu à faire face à certaines particularités de Salome TMF comme le fait que Salome Tmf nécessite l'installation d'une ancienne version de java (version JRE 1.6.17), que le plugin Mantis nécessite l'installation du programme mantis.jar particulier car celui de l'installation d'origine comportait des erreurs. Mais mon stage a été très intéressant de ce point de vue car j'ai du installer de nombreux outils notamment TOPCASED, WAMP (Php, Mysql, Apache), les inclure dans SALOME TMF et ses nombreux plugins, DRESDEN OCL TOOLKIT et apprendre à les utiliser. Je peux dire que je maitrise maintenant tous ces outils et même les nuances du langage OCL.

Cependant malgré mes difficultés et notamment le handicap d'avoir une expérience informatique ancienne, j'ai toujours montré de la persévérance et de la perspicacité, et un grand sens de l'organisation pour résoudre les problèmes, ceci m'a permis d'atteindre le niveau que j'ai aujourd'hui.

Enfin j'ai découvert le monde de la recherche, constitué de gens sympathiques, ouverts, passionnés par leur métier qui travaillent en étroite collaboration avec l'industrie et en même temps qui la précèdent. Ceci m'a permis de découvrir des outils avant-gardistes, dans une ambiance pacifique et créatrice de résultats concrets dont certains sont directement applicables à l'industrie contrairement à l'image habituellement véhiculée sur le monde de la recherche.

BIBLIOGRAPHIE

Articles :

[CEG08] CLAVEL M.,EGEA M.,A.GARCIA DE DIOS M.,2008,Building an Efficient Component for OCL Evaluation

[Com09] COMBEMALE B.,2009, Ingénierie dirigée par les modèles (IDM), état de l'art hal-00371565,version 1- 29 mars 2009

[CPP08] CHIOREAN D.,PETRASCU V.,PETRASCU D., 2008, How My Favorite Tool Supporting OCL Must Look Like

[GKB08a] GOGOLLA M.,KUHLMANN M. ,BUTTNER F., 2008, Source for a Benchmark for OCL Engine Accuracy, Determinateness, and Efficiency

[GKB08b] GOGOLLA M.,KUHLMANN M. ,BUTTNER F., 2008, A Benchmark for OCL Engine Accuracy, Determinateness, and Efficiency

[GK07] GOGOLLA M.,KUHLMANN M, 2007, Analyzing Semantic Properties of OCL Operations by Uncovering Interoperational Relationships. In: Electronic Communications of the EASST, UML/MoDELS Workshop on OCL (OCL4ALL 2007) , vol. 9, pages 17

[TRF03] TOVAL A., REQUENA V., FERNANDEZ J.L.,2003, Emerging OCL tools

[WTF11] WILKE C., THIELE M., FREITAG B.,2011, Dresden OCL manual for installation, use and development

Ouvrages :

[DD02] DEITEL H .M.,DEITEL P.J., 2002, Comment Programmer en Java Quatrième édition, Edition Reynald Goulet inc.

[Gar02] GARDARIN G., 2002, XML Des bases de données aux services Web, Dunod, Paris

[Lav10] LAVERGNE G.,2010, Etude de Salome TMF et comparaison avec d'autres outils de tests

[Mcl02] MCLAUGBLIN B., 2002, Java et Xml, O'REILLY & Associates

[MVM10] MILLER F .P., VANDOME A.F.,MACBREWSTER J., 2010, BeanShell

[OMG09] OMG Available Specification 2009 Object Constraint Langage 2.1

[Tid08] TIDWELL D., 2008, Xslt ,O'REILLY & Associates

ANNEXES

REGLES OCL

MAIL DE M. GABEL

MODE D'EMPLOI SALOME TMF

FICHIER PLANTAGECOMPIL_TOPCASED.TXT

FICHIER RESULTAT DE NEPTUNE

FICHIER RESULTAT DE TOPCASED

FICHIER RESULTAT COMPARAISON DE TOPCASED/NEPTUNE

MAIL DE M. GABEL

Le 08/11/2011 14:37, Sébastien GABEL a écrit :

Bonjour,

>>

Le problème est certainement que TOPCASED ne trouve pas leméta-modèle auquel se réfère le modèle My.personnes.

Le checker OCL de TOPCASED utilise le registre EMF pour chercher le méta-modèle d'un modèle donné à l'aide de son URI (ce n'est pas le cas ici, on passe un chemin vers un MM en vue de faire un chargement dynamique). Ceci implique, au préalable, d'avoir généré la couche EMF et de l'avoir déployé au sein de l'atelier en ce qui concerne le méta-modèle.

Il vous faut impérativement réaliser cette étape (au moins générer la couche modèle du MM).

Typiquement, une des limitations de TOPCASED est qu'il ne sait pas traiter ce genre de cas vu au sein de votre modèle et avec du recul, c'est fort dommage :

xmlns:My="platform:/resource/mmPersonne/model/My.ecore"

par contre, il s'en sortirait bien avec quelque chose comme xmlns:My=http://My.ecore/1.0 qui serait connu et chargé au sein du registre EMF.

Mode d’emploi

SALOME TMF

Test Management Framework

TABLE DES MATIERES SALOME TMF

I-INSTALLATION

Prérequis :

installer wampserver 2.1**,** *lien* :

http://www.01net.com/telecharger/windows/Internet/editeur_de_site/fiches/28739.html

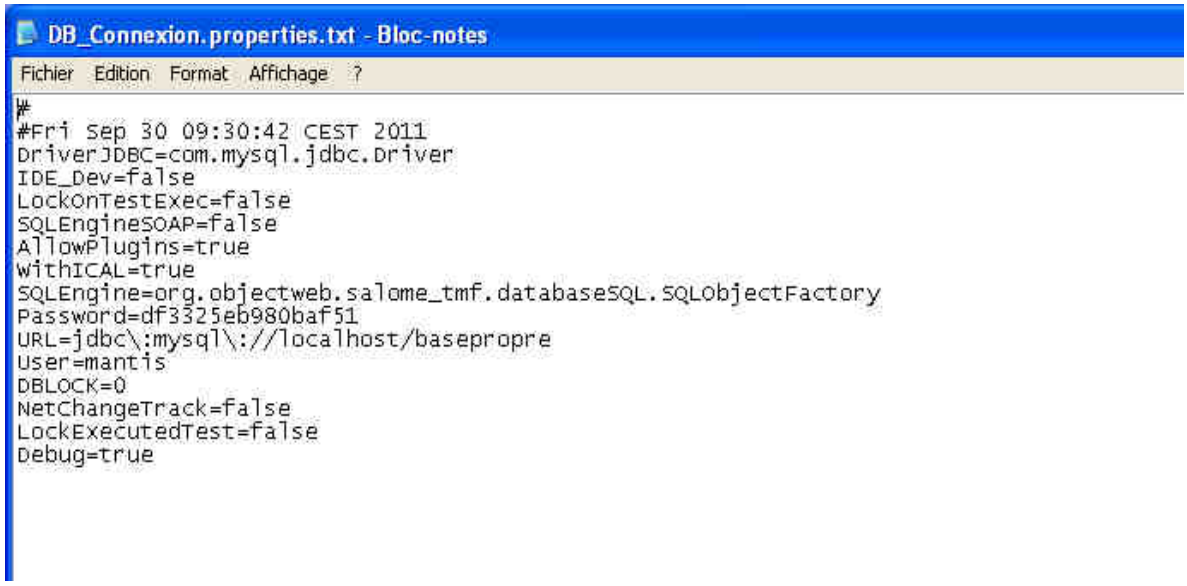
Se référer à la documentation SALOME TMF pour l'installation et l'utilisation de SALOME TMF auteur : Mickaël Marche date : 26 avril 2006 qui se trouve dans « C:\Program files\SalomeTMF\docs\pdf\fr ».
mais :

Problèmes rencontrés et résolution :

SALOME TMF version 3 (à utiliser car chargement automatique) ne fonctionne qu'avec la version JAVA :jdk1.6.0_17.
Lien pour installation automatique de SALOME TMF sous le répertoire « c:\wamp\www\ »:

http://www.commentcamarche.net/faq/16793-installation-salome-tmf#installation

Le fichier DB_Connexion.properties du répertoire « C:\wamp\www\salome_tmf\cfg » doit être de la forme :



- Database name est « basepropre »
- User name est « mantis »
- Et le password est cripté

Toutes ces données sont rentrées grâce à « **l'outil de configuration de la base de données** ».voir page 12 de la documentation de Salome TMF de Mickaël Marche.

Deuxième étape : le fichier create_salome_bdd.sql qui se trouve dans le répertoire« C:\wamp\www\salome_tmf\bdd_model » permet de créer toutes les tables

Salome TMF. Il est généré automatiquement après avoir rempli l'écran de saisie « **Paramètres de création d'une base de données Salome TMF** »

LIENS INDISPENSABLES POUR SALOME TMF/

Administration de Salome TMF (créer un projet et un utilisateur)

http://www.commentcamarche.net/faq/16863-administration-de-salome-tmf

Utilisation des plugins de Salome TMF

http://www.commentcamarche.net/faq/16875-utilisation-avancee-de-salome-tmf-plugins#installation-des-plugins

Différentes versions des plugins de Salome TMF

http://forge.ow2.org/project/showfiles.php?group_id=194

PLUGINS SALOME A INSTALLER doivent avoir une version antérieure à la version de Salome TMF qui est la version 3 qui date de 20 décembre 2007

Beanshell-3 qui date du 29/11/2007.
cronExec-3 qui date du 29/11/2007
docXML-3 qui date du 20/12/2007
helpgui-3 qui date du 29/11/2007
mantis-3 qui date du 20/12/2007

Important : Le fichier mantis.jar n'est pas bon, le fichier correct est à télécharger à l'adresse suivante : http://forge.ow2.org/project/download.php?group_id=194&file_id=9757 **sinon il affiche le message « Mantis ne peut pas charger la configuration définie dans votre fichier de configuration »**

requirements-3 qui date du 20/12/2007
simplejunit-3 qui date du 20/12/2007

Pour installer les plugins de Salome TMF se référer à la documentation de Salome TMF de Mickaël Marche page 15

Remarque : J'ai rencontré des problèmes de fichiers caches du navigateur qui empêchaient la bonne installation des plugins, si tel est le cas, il faut les purger.

POUR CREER UN PROJET ET DES UTLISATEUR DANS SALOME

voir la documentation de SALOME TMF auteur : Mickaël Marche pages16 à 25.

POUR CREER UNE CAMPAGNE DE TEST, UNE FAMILLE,UNE SUITE ET DES TESTS UN ENVIRONNEMENT voir la documentation de salome_tmf de Mickaël r pages 27 à 51

II-PROGRAMMATION

Pour programmer Salome TMF c'est-à-dire ajouter un script de test, un script d'environnement et exécuter en batch le programme que je veux tester, j'ai trouvé le *lien* suivant
: http://forge.ow2.org/forum/forum.php?thread_id=2926&forum_id=867

Il contient un script Beanshell qui permet de lancer un test dans Salome TMFet indique les grandes lignes pour faire fonctionner Salome TMF.

III-MARCHE A SUIVRE POUR AUTOMATISER

La première étape pour automatiser est la création d'un environnement de test.

Comment **créer un environnement.**
Dans « Gestion des données » (à côté de l'onglet gestion des campagnes). Cliquer sur l'onglet « Gestion des données » cliquer sur « Environnement » et sur « Ajouter », renseigner ensuite le nom de l'environnement », ici nous voulons tester le logiciel Topcased donc dans le nom de l'environnement nous tapons « Topcased ».
Cliquer ensuite sur « Chercher » en dessous de nom du script, puis sélectionner le fichier d'extension Java, ici testtopcased.java qui est le programme java permettant de lancer notre environnement topcased et ainsi de lancer notre programme batch dont nous voulons tester les résultats dans Salome TMF ce qui produit en sortie un fichier XMI.

Créer ensuite le paramètre TEST_DRIVER_NAME qui contient le nom de l'exécutable (le « .exe » associé à notre programme Java ici testtopcased.exe

Pour obtenir ce fichier nous avons utilisé la logiciel JavaExe qui permet de créer un exécutable « .exe » à partir d'un fichier d'extention « .class » : testtopcased.class (obtenu par compilation du fichier testtopcased.java). Il nous a suffit pour cela de copier le fichier JavaExe_console dans le répertoire du fichier testopcased.java et de le renommer en testtopcased.

Cliquer ensuite sur « Attachment », et **rattacher à l'environnement tous les fichiers** d'extension « java », « class », « exe » et les fichiers « jar » et « bat » **nécessaires à l'évaluation en batch** des requêtes OCL dans Topcased.

Pour cela, cliquer sur l'onglet « Attachment » puis sur « Ajouter fichier » et sélectionner le fichier « .exe » ici testtopcased.exe, et tous les autres fichiers nécessaires à l'exécution en batch puis valider.

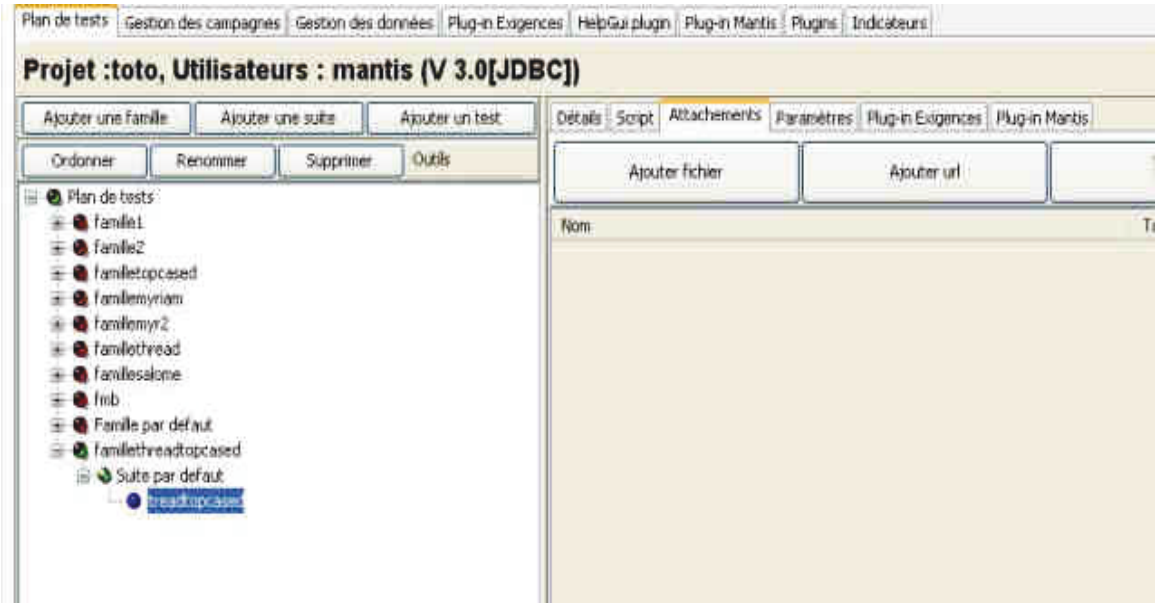
Cliquer ensuite sur « Valider » : notre environnement de test est maintenant créé.

Après avoir créé un environnement et le paramètre TEST_DRIVER_NAME

Créer une famille pour cela :
Cliquer sur l'onglet « Plan des tests »

- puis « Ajouter une famille »,
- puis « Ajouter une suite »
- puis « Ajouter un test »

Après avoir créé un test Beanshell (voir pages 29 et 30 de SALOME TMF auteur : Mickaël Marche). Page 30 fenêtre « Nouveau Test », cliquer sur la liste déroulante du type de test par défaut « Manuel » et choisir « Beanshell ». Cliquer ensuite sur le test créé, ici « testtopcased »



Puis rattacher le fichier d'extension Beanshell préalablement créé au test.

Pour cela,

- cliquez ensuite sur l'onglet « Script » : le deuxième onglet sur la partie droite de l'écran ci-dessus,
- cliquer sur « Ajouter »,
- puis cliquer ensuite sur « Script Existant »,
- et sur « Browse »
- et choisir le fichier Beanshell d'extension « .bsh », **ici testtopcased.bsh** dont un exemple figure ci-après. Les scripts Beanshell n'ont pas besoin d'être compilés car ils sont interprétés.
- Il reste à associer notre paramètre TEST_DRIVER_NAME au test pour cela cliquer dans « Paramètres » et « Utiliser » et choisir le paramètre à utiliser ici TEST_DRIVER_NAME

Détail du script Beanshell d'initialisation testtopcased.bsh :

```
import org.objectweb.salome_tmf.api.*;
import org.objectweb.salome_tmf.api.api2ihm.adminProject.*;
import org.objectweb.salome_tmf.api.api2ihm.campTest.*;
import org.objectweb.salome_tmf.api.api2ihm.suiteTest.*;
import org.objectweb.salome_tmf.api.api2ihm.adminVT.*;
import org.objectweb.salome_tmf.data.*;
import salomeTMF_plug.beanshell.ParentClassLoader;

import java.awt.image.BufferedImage;
import java.io.File;
import java.net.URL;

/*init*/
Verdict = "fail";
Runtime salomeRunTime = Runtime.getRuntime();

try {
      /*Download du driver de test*/
      execFileAttachement =
         salome_environmentObject.getAttachmentFromModel(TEST_DRIVER_NAME);
      testLog = testLog + "Attachement is " + execFileAttachement + "\n";
      File testtopcasedFile ;
      if (execFileAttachement.isInBase())
      {
            testtopcasedFile = execFileAttachement.getFileFromDB();
      }
      else
      {
            testtopcasedFile = execFileAttachement.getLocalFile();
      }
      testtopcasedPath = testtopcasedFile.getPath();

      testLog = "Download du driver de test\n";
      testLog = testLog + "Name : " + testtopcasedFile.getName() + "\n";
      testLog = testLog + "Path : " + testtopcasedFile.getPath() + "\n";

      testLog = testLog + "\nExecution du driver de test\n";
      if (File.separator.equals("/"))
      {
            /* Sous Unix/Linux rendre le fichier executable */
            salomeRunTime.exec("chmod 755 " +  testtopcasedPath);
      }
      else
      {
            execCommand = testtopcasedPath;
      }

      pProcess = salomeRunTime.exec(execCommand);
      pProcess.waitFor();

      int exitValue = pProcess.exitValue();
      testLog = testLog + "exitValue : " + exitValue +  "\n";
      if (exitValue == 0){
            Verdict = "pass";
      }

      /*Attacher le fichier de log produit par outil externe*/
```

```
      /* On suppose ici que testtopcased produit un fichier de résulat */
      /* TEST_DRIVER_NAME.log dans le tmp */
      /* Si c'est le cas on rajoute ce fichier en attachement du */
      /* resultat d'execution du test */
      String tmpDir = System.getProperties().getProperty("java.io.tmpdir");
      String fs = System.getProperties().getProperty("file.separator");
      File fileLog = new File(tmpDir + fs + TEST_DRIVER_NAME +".log");
      if (fileLog.exists())
      {
            addTestResultAttachment (fileLog);
      }

}
catch (Exception e)
{
      testLog = testLog + e.toString()+e.printStackTrace();
}
```

Détail de testtopcased.java :

```
import java.io.*;
public class testtopcased {
public static void main(String[] args)
{
      try
      {
            String[] command = { "C:\\Documents and
Settings\\ballarin\\Bureau\\Topcased-RCP-win32-4.3.0\\Topcased-
4.3.0\\eclipsec.exe" ,
                  "-Xmx1024m", "-clean", "-nosplash",
                  "-data",
                  "F:\\ballarin\\Mes documents\\workspace",
                  "-application", "org.topcased.ocl.batch.checkfile",
                  "-model",
                  "F:\\ballarin\\Mes documents\\runtime-
EclipseApplication\\mmPersonneOcl\\Cec.gens",
                  "-rule",
                  "F:\\ballarin\\Mes documents\\runtime-
EclipseApplication\\mmPersonneOcl\\ocl_rule1.ocl",
                  "-output",
                  "F:\\ballarin\\Mes documents\\runtime-
EclipseApplication\\mmPersonneOcl\\results",
                  "-logmodel", "respersonneocl.txt"} ;

            Runtime r = Runtime.getRuntime();
            Process p = r.exec(command);
            p.waitFor();
            System.exit(0);
      }
      catch (Exception e)
      {
            System.out.println(e.getMessage());
            System.exit(-1);
      }
   }
}
```

Il est à noter que nous exécutons ici les règles OCL qui sont dans le fichier :
« F:\ballarin\Mes documents\runtime-EclipseApplication\mmPersonneOcl\ocl_rule1.ocl »

Créer ensuite une nouvelle campagne en important la famille de test créée préalablement avec l'onglet « Ajouter une famille », testtopcased que nous avons créé précédemment. Pour cela cliquer sur la campagne et cliquer sur le bouton « importer » et importer les tests que l'on veut exécuter dans notre campagne.

Puis lancer une exécution.

IMPORTANT* : *ne pas oublier de copier dans notre répertoire TEMP de Salome_data (F:\ballarin\TEMP\salome_data) le fichier d'extension class ici testtopcased.class

Pour cela cliquer sur la campagne de test que l'on veut lancer qui se trouve dans l'onglet « Campagne de test » puis sur l'onglet « Exécution » en haut à droite de l'écran puis sur « Ajouter »
Dans la fenêtre « Nouvelle exécution » **sélectionner l'environnement** « Topcased » que l'on a créé, pour cela dans le cadre « Environnement » cliquer sur la liste déroulante et sélectionner notre environnement.

IMPORTANT : Lors du lancement d'une exécution, il faut rattacher un script d'initialisation à l'exécution :

Après avoir sélectionné l'environnement ici « Topcased » pour l'exécution,
Cliquer sur « Ajouter » en dessous de « Script d'initialisation »
Sélectionner le script Beanshell ici « testtopcased.bsh » et valider
Cliquer ensuite sur valider.

Notre exécution « Exec0 » apparait, cliquer sur l'onglet de démarrage programmé :

Dans le cadre « Exécution » cliquer sur l'exécution que l'on veut lancer puis sur la touche « > », puis renseigner l'heure de lancement (pour faire un lancement différé) puis valider.

Cliquer ensuite sur « Exec0 », les résultats d'exécution s'affichent au dessous dans le cadre résultat, pour voir le détail cliquer dessus puis sur « Détail». Si dans la colonne résultat il y a une coche entourée d'un rond vert çà veut dire que le résultat du test est ok, cliquer dessus puis sur « Détail », puis cliquer sur le fichier résultat et sur « Visualiser », apparait alors le déroulement de l'exécution avec le code retour de l'exécution. Notre fichier résultat de Topcased se trouve dans

```
F:\ballarin\Mes documents\runtime-EclipseApplication\mmPersonneOcl\results
```

IV-COMMENT EDITER UN COMPTE RENDU DU RESULTAT DES TESTS

Le plugin GENDOC de SALOME TMF offre notamment la possibilité de créer des rapports sur les plans de tests, les campagnes.
L'accès aux fonctionnalités du plugin GENDOC se fait via le bouton « **Outils** », présents dans les trois premiers onglets de SALOME TMF : « Gestion des tests », « Gestion des campagnes » et « Gestion des données ».
Pour générer le rapport des campagnes de test, se positionner sur l'onglet « Gestion des campagnes » puis cliquer sur « Outils » à côté du bouton « Créer une campagne », choisir l'option « Plugin génération de documents » et choisir entre « Dossier de test » et « Résultats des tests ». Choisir ensuite la directory ou stocker le rapport et modifier si besoin le nom du rapport qui est par défaut « index.html ».

FICHIER PLANTAGECOMPIL_TOPCASED.TXT

```
let ada:Personne=Personne.allInstances()->any(p:Personne|p.name='Ada') in
let emptySeq=Sequence{ada}->excluding(ada) in
Personne.allInstances()->iterate(w,h:Person;

res:Bag(Sequence(Person))=Bag{emptySeq}->excluding(emptySeq)|                    if
w.gender=Gender::female and w.alive and w.civstat<>Civstatus::married and

  h.gender=Gender::male  and  h.alive  and  h.civstat<>Civstatus::married  then  res-
>including(Sequence{w,h})

 else  res  endif)->  collect(pair:Sequence(Person)|pair->collect(p:Personne|p.name))=
Bag{Sequence{'Ada','Bob'}}
```

Parsing Exception : 316:75:316:77 "let" unexpected token(s)

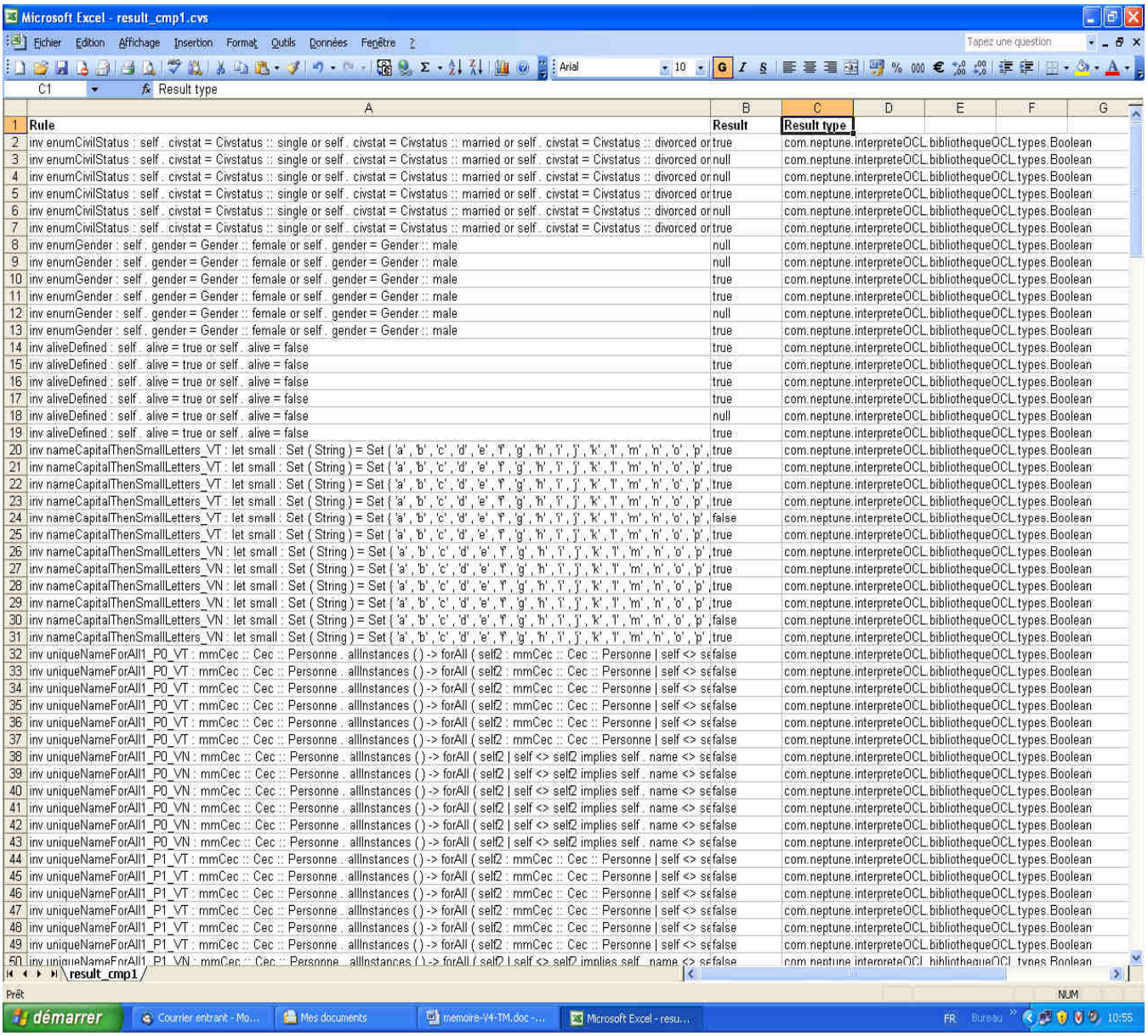
```
context gens::Personne inv possiblePairs_P0_VT:
let ada:Personne=Personne.allInstances()->any(p:Personne|p.name='Ada') in
let emptySeq:Personne=Sequence{ada}:Personne->excluding(ada) in
Personne.allInstances()->iterate(w,h:Person;

res:Bag(Sequence(Personne))=Bag{emptySeq}->excluding(emptySeq)|                  if
w.gender=Gender::female and w.alive and w.civstat<>Civstatus::married and

  h.gender=Gender::male  and  h.alive  and  h.civstat<>Civstatus::married  then  res-
>including(Sequence{w,h})

 else  res  endif)->  collect(pair:Sequence(Personne)|pair->collect(p:Personne|p.name))=
Bag{Sequence{'Ada','Bob'}}
```

Parsing Exception : 315:110:315:110 "." expected instead of ":"

```
context gens::Personne inv abcAttrs_P0_VT:

let ada:Personne=Personne.allInstances()->any(p:Personne|p.name='Ada') in
let bob:Personne=Personne.allInstances()->any(p:Personne|p.name='Bob') in
let cyd:Personne=Personne.allInstances()->any(p:Personne|p.name='Cyd') in
Set{ada,bob,cyd}->

  collect(p:Personne|Sequence{p.name,p.civstat,p.gender,p.alive})=
Bag{Sequence{'Ada',Civstatus::widowed ,Gender::female,true },

Sequence{'Bob',Civstatus::divorced,Gender::male ,true }, Sequence{'Cyd',Civstatus::married
,Gender::male ,false}}
```

Parsing Exception : Cannot find operation (=(Bag(Sequence(OclAny)))) for the type (Bag(OclAny)) -- error location not published by the parser

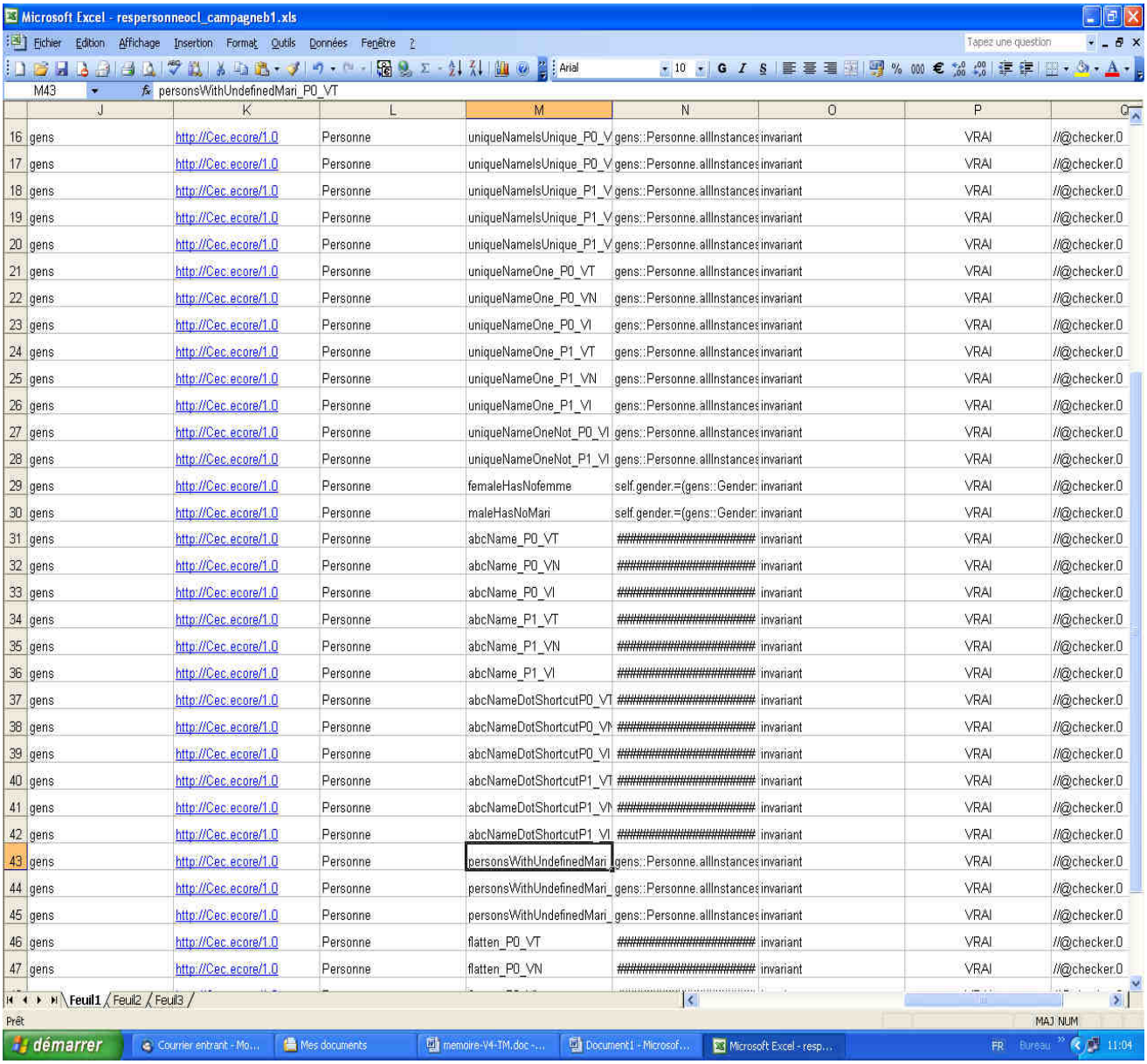
FICHIER RESULTAT DE NEPTUNE



	Rule	Result	Result type
2	inv enumCivilStatus : self . civstat = Civstatus :: single or self . civstat = Civstatus :: married or self . civstat = Civstatus :: divorced or	true	com.neptune.interpreteOCL.bibliothequeOCL.types.Boolean
3	inv enumCivilStatus : self . civstat = Civstatus :: single or self . civstat = Civstatus :: married or self . civstat = Civstatus :: divorced or	null	com.neptune.interpreteOCL.bibliothequeOCL.types.Boolean
4	inv enumCivilStatus : self . civstat = Civstatus :: single or self . civstat = Civstatus :: married or self . civstat = Civstatus :: divorced or	null	com.neptune.interpreteOCL.bibliothequeOCL.types.Boolean
5	inv enumCivilStatus : self . civstat = Civstatus :: single or self . civstat = Civstatus :: married or self . civstat = Civstatus :: divorced or	true	com.neptune.interpreteOCL.bibliothequeOCL.types.Boolean
6	inv enumCivilStatus : self . civstat = Civstatus :: single or self . civstat = Civstatus :: married or self . civstat = Civstatus :: divorced or	null	com.neptune.interpreteOCL.bibliothequeOCL.types.Boolean
7	inv enumCivilStatus : self . civstat = Civstatus :: single or self . civstat = Civstatus :: married or self . civstat = Civstatus :: divorced or	true	com.neptune.interpreteOCL.bibliothequeOCL.types.Boolean
8	inv enumGender : self . gender = Gender :: female or self . gender = Gender :: male	null	com.neptune.interpreteOCL.bibliothequeOCL.types.Boolean
9	inv enumGender : self . gender = Gender :: female or self . gender = Gender :: male	null	com.neptune.interpreteOCL.bibliothequeOCL.types.Boolean
10	inv enumGender : self . gender = Gender :: female or self . gender = Gender :: male	true	com.neptune.interpreteOCL.bibliothequeOCL.types.Boolean
11	inv enumGender : self . gender = Gender :: female or self . gender = Gender :: male	true	com.neptune.interpreteOCL.bibliothequeOCL.types.Boolean
12	inv enumGender : self . gender = Gender :: female or self . gender = Gender :: male	null	com.neptune.interpreteOCL.bibliothequeOCL.types.Boolean
13	inv enumGender : self . gender = Gender :: female or self . gender = Gender :: male	true	com.neptune.interpreteOCL.bibliothequeOCL.types.Boolean
14	inv aliveDefined : self . alive = true or self . alive = false	true	com.neptune.interpreteOCL.bibliothequeOCL.types.Boolean
15	inv aliveDefined : self . alive = true or self . alive = false	true	com.neptune.interpreteOCL.bibliothequeOCL.types.Boolean
16	inv aliveDefined : self . alive = true or self . alive = false	true	com.neptune.interpreteOCL.bibliothequeOCL.types.Boolean
17	inv aliveDefined : self . alive = true or self . alive = false	true	com.neptune.interpreteOCL.bibliothequeOCL.types.Boolean
18	inv aliveDefined : self . alive = true or self . alive = false	null	com.neptune.interpreteOCL.bibliothequeOCL.types.Boolean
19	inv aliveDefined : self . alive = true or self . alive = false	true	com.neptune.interpreteOCL.bibliothequeOCL.types.Boolean
20	inv nameCapitalThenSmallLetters_VT : let small : Set (String) = Set { 'a' , 'b' , 'c' , 'd' , 'e' , 'f' , 'g' , 'h' , 'i' , 'j' , 'k' , 'l' , 'm' , 'n' , 'o' , 'p' ,	true	com.neptune.interpreteOCL.bibliothequeOCL.types.Boolean
21	inv nameCapitalThenSmallLetters_VT : let small : Set (String) = Set { 'a' , 'b' , 'c' , 'd' , 'e' , 'f' , 'g' , 'h' , 'i' , 'j' , 'k' , 'l' , 'm' , 'n' , 'o' , 'p' ,	true	com.neptune.interpreteOCL.bibliothequeOCL.types.Boolean
22	inv nameCapitalThenSmallLetters_VT : let small : Set (String) = Set { 'a' , 'b' , 'c' , 'd' , 'e' , 'f' , 'g' , 'h' , 'i' , 'j' , 'k' , 'l' , 'm' , 'n' , 'o' , 'p' ,	true	com.neptune.interpreteOCL.bibliothequeOCL.types.Boolean
23	inv nameCapitalThenSmallLetters_VT : let small : Set (String) = Set { 'a' , 'b' , 'c' , 'd' , 'e' , 'f' , 'g' , 'h' , 'i' , 'j' , 'k' , 'l' , 'm' , 'n' , 'o' , 'p' ,	true	com.neptune.interpreteOCL.bibliothequeOCL.types.Boolean
24	inv nameCapitalThenSmallLetters_VT : let small : Set (String) = Set { 'a' , 'b' , 'c' , 'd' , 'e' , 'f' , 'g' , 'h' , 'i' , 'j' , 'k' , 'l' , 'm' , 'n' , 'o' , 'p' ,	false	com.neptune.interpreteOCL.bibliothequeOCL.types.Boolean
25	inv nameCapitalThenSmallLetters_VT : let small : Set (String) = Set { 'a' , 'b' , 'c' , 'd' , 'e' , 'f' , 'g' , 'h' , 'i' , 'j' , 'k' , 'l' , 'm' , 'n' , 'o' , 'p' ,	true	com.neptune.interpreteOCL.bibliothequeOCL.types.Boolean
26	inv nameCapitalThenSmallLetters_VN : let small : Set (String) = Set { 'a' , 'b' , 'c' , 'd' , 'e' , 'f' , 'g' , 'h' , 'i' , 'j' , 'k' , 'l' , 'm' , 'n' , 'o' , 'p' ,	true	com.neptune.interpreteOCL.bibliothequeOCL.types.Boolean
27	inv nameCapitalThenSmallLetters_VN : let small : Set (String) = Set { 'a' , 'b' , 'c' , 'd' , 'e' , 'f' , 'g' , 'h' , 'i' , 'j' , 'k' , 'l' , 'm' , 'n' , 'o' , 'p' ,	true	com.neptune.interpreteOCL.bibliothequeOCL.types.Boolean
28	inv nameCapitalThenSmallLetters_VN : let small : Set (String) = Set { 'a' , 'b' , 'c' , 'd' , 'e' , 'f' , 'g' , 'h' , 'i' , 'j' , 'k' , 'l' , 'm' , 'n' , 'o' , 'p' ,	true	com.neptune.interpreteOCL.bibliothequeOCL.types.Boolean
29	inv nameCapitalThenSmallLetters_VN : let small : Set (String) = Set { 'a' , 'b' , 'c' , 'd' , 'e' , 'f' , 'g' , 'h' , 'i' , 'j' , 'k' , 'l' , 'm' , 'n' , 'o' , 'p' ,	true	com.neptune.interpreteOCL.bibliothequeOCL.types.Boolean
30	inv nameCapitalThenSmallLetters_VN : let small : Set (String) = Set { 'a' , 'b' , 'c' , 'd' , 'e' , 'f' , 'g' , 'h' , 'i' , 'j' , 'k' , 'l' , 'm' , 'n' , 'o' , 'p' ,	false	com.neptune.interpreteOCL.bibliothequeOCL.types.Boolean
31	inv nameCapitalThenSmallLetters_VN : let small : Set (String) = Set { 'a' , 'b' , 'c' , 'd' , 'e' , 'f' , 'g' , 'h' , 'i' , 'j' , 'k' , 'l' , 'm' , 'n' , 'o' , 'p' ,	true	com.neptune.interpreteOCL.bibliothequeOCL.types.Boolean
32	inv uniqueNameForAll1_P0_VT : mmCec :: Cec :: Personne . allInstances ( ) -> forAll (self2 : mmCec :: Cec :: Personne | self <> se	false	com.neptune.interpreteOCL.bibliothequeOCL.types.Boolean
33	inv uniqueNameForAll1_P0_VT : mmCec :: Cec :: Personne . allInstances ( ) -> forAll (self2 : mmCec :: Cec :: Personne | self <> se	false	com.neptune.interpreteOCL.bibliothequeOCL.types.Boolean
34	inv uniqueNameForAll1_P0_VT : mmCec :: Cec :: Personne . allInstances ( ) -> forAll (self2 : mmCec :: Cec :: Personne | self <> se	false	com.neptune.interpreteOCL.bibliothequeOCL.types.Boolean
35	inv uniqueNameForAll1_P0_VT : mmCec :: Cec :: Personne . allInstances ( ) -> forAll (self2 : mmCec :: Cec :: Personne | self <> se	false	com.neptune.interpreteOCL.bibliothequeOCL.types.Boolean
36	inv uniqueNameForAll1_P0_VT : mmCec :: Cec :: Personne . allInstances ( ) -> forAll (self2 : mmCec :: Cec :: Personne | self <> se	false	com.neptune.interpreteOCL.bibliothequeOCL.types.Boolean
37	inv uniqueNameForAll1_P0_VT : mmCec :: Cec :: Personne . allInstances ( ) -> forAll (self2 : mmCec :: Cec :: Personne | self <> se	false	com.neptune.interpreteOCL.bibliothequeOCL.types.Boolean
38	inv uniqueNameForAll1_P0_VN : mmCec :: Cec :: Personne . allInstances ( ) -> forAll (self2 | self <> self2 implies self . name <> se	false	com.neptune.interpreteOCL.bibliothequeOCL.types.Boolean
39	inv uniqueNameForAll1_P0_VN : mmCec :: Cec :: Personne . allInstances ( ) -> forAll (self2 | self <> self2 implies self . name <> se	false	com.neptune.interpreteOCL.bibliothequeOCL.types.Boolean
40	inv uniqueNameForAll1_P0_VN : mmCec :: Cec :: Personne . allInstances ( ) -> forAll (self2 | self <> self2 implies self . name <> se	false	com.neptune.interpreteOCL.bibliothequeOCL.types.Boolean
41	inv uniqueNameForAll1_P0_VN : mmCec :: Cec :: Personne . allInstances ( ) -> forAll (self2 | self <> self2 implies self . name <> se	false	com.neptune.interpreteOCL.bibliothequeOCL.types.Boolean
42	inv uniqueNameForAll1_P0_VN : mmCec :: Cec :: Personne . allInstances ( ) -> forAll (self2 | self <> self2 implies self . name <> se	false	com.neptune.interpreteOCL.bibliothequeOCL.types.Boolean
43	inv uniqueNameForAll1_P0_VN : mmCec :: Cec :: Personne . allInstances ( ) -> forAll (self2 | self <> self2 implies self . name <> se	false	com.neptune.interpreteOCL.bibliothequeOCL.types.Boolean
44	inv uniqueNameForAll1_P1_VT : mmCec :: Cec :: Personne . allInstances ( ) -> forAll (self2 : mmCec :: Cec :: Personne | self <> se	false	com.neptune.interpreteOCL.bibliothequeOCL.types.Boolean
45	inv uniqueNameForAll1_P1_VT : mmCec :: Cec :: Personne . allInstances ( ) -> forAll (self2 : mmCec :: Cec :: Personne | self <> se	false	com.neptune.interpreteOCL.bibliothequeOCL.types.Boolean
46	inv uniqueNameForAll1_P1_VT : mmCec :: Cec :: Personne . allInstances ( ) -> forAll (self2 : mmCec :: Cec :: Personne | self <> se	false	com.neptune.interpreteOCL.bibliothequeOCL.types.Boolean
47	inv uniqueNameForAll1_P1_VT : mmCec :: Cec :: Personne . allInstances ( ) -> forAll (self2 : mmCec :: Cec :: Personne | self <> se	false	com.neptune.interpreteOCL.bibliothequeOCL.types.Boolean
48	inv uniqueNameForAll1_P1_VT : mmCec :: Cec :: Personne . allInstances ( ) -> forAll (self2 : mmCec :: Cec :: Personne | self <> se	false	com.neptune.interpreteOCL.bibliothequeOCL.types.Boolean
49	inv uniqueNameForAll1_P1_VT : mmCec :: Cec :: Personne . allInstances ( ) -> forAll (self2 : mmCec :: Cec :: Personne | self <> se	false	com.neptune.interpreteOCL.bibliothequeOCL.types.Boolean
50	inv uniqueNameForAll1_P1_VN : mmCec :: Cec :: Personne . allInstances ( ) -> forAll (self2 | self <> self2 implies self . name <> se	false	com.neptune.interpreteOCL.bibliothequeOCL.types.Boolean

Fig. 1. Détail Fichier Neptune

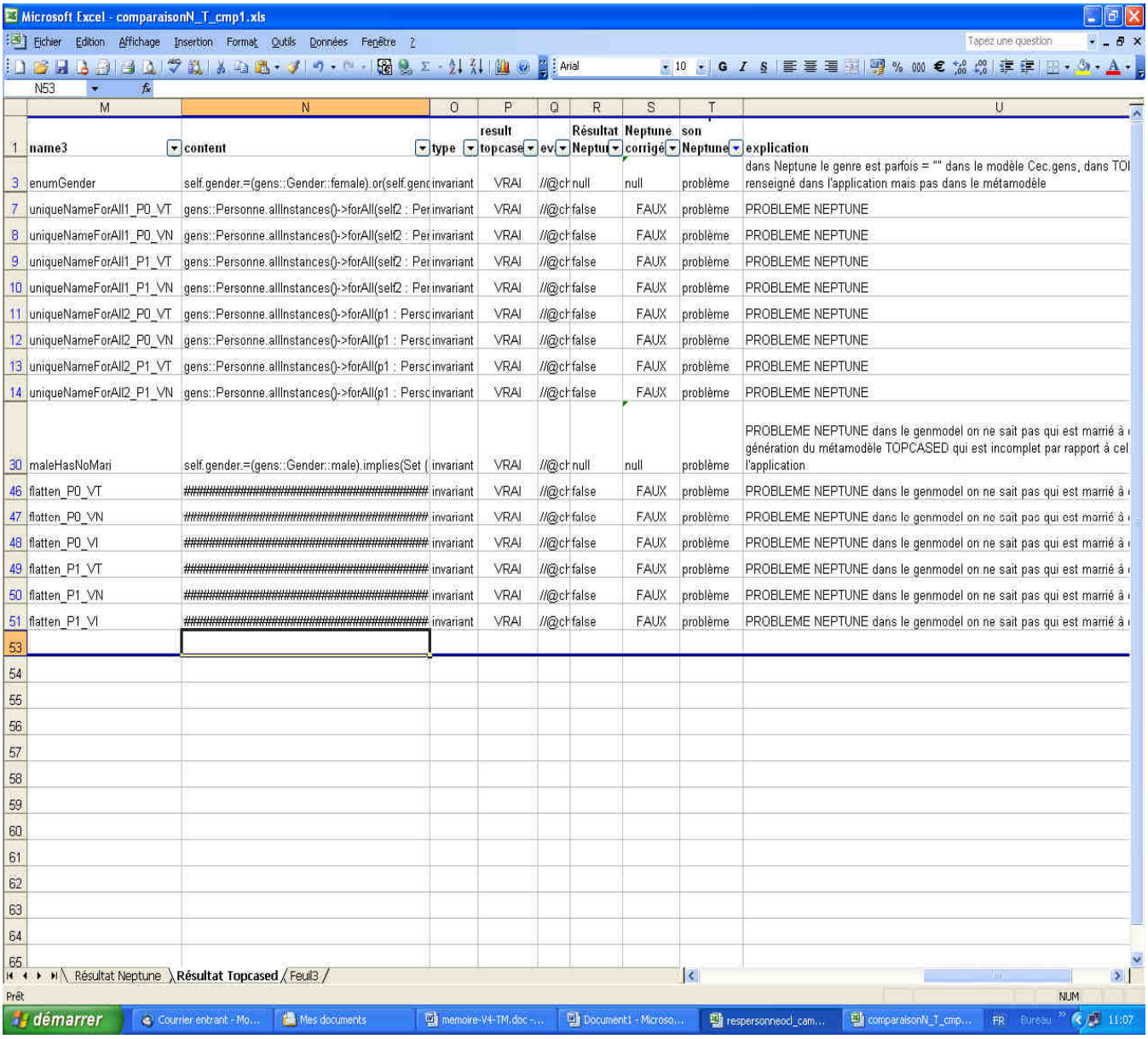
FICHIER RESULTAT DE TOPCASED



	J	K	L	M	N	O	P	Q
16	gens	http://Cec.ecore/1.0	Personne	uniqueNameIsUnique_P0_V	gens::Personne.allInstances	invariant	VRAI	//@checker.0
17	gens	http://Cec.ecore/1.0	Personne	uniqueNameIsUnique_P0_V	gens::Personne.allInstances	invariant	VRAI	//@checker.0
18	gens	http://Cec.ecore/1.0	Personne	uniqueNameIsUnique_P1_V	gens::Personne.allInstances	invariant	VRAI	//@checker.0
19	gens	http://Cec.ecore/1.0	Personne	uniqueNameIsUnique_P1_V	gens::Personne.allInstances	invariant	VRAI	//@checker.0
20	gens	http://Cec.ecore/1.0	Personne	uniqueNameIsUnique_P1_V	gens::Personne.allInstances	invariant	VRAI	//@checker.0
21	gens	http://Cec.ecore/1.0	Personne	uniqueNameOne_P0_VT	gens::Personne.allInstances	invariant	VRAI	//@checker.0
22	gens	http://Cec.ecore/1.0	Personne	uniqueNameOne_P0_VN	gens::Personne.allInstances	invariant	VRAI	//@checker.0
23	gens	http://Cec.ecore/1.0	Personne	uniqueNameOne_P0_VI	gens::Personne.allInstances	invariant	VRAI	//@checker.0
24	gens	http://Cec.ecore/1.0	Personne	uniqueNameOne_P1_VT	gens::Personne.allInstances	invariant	VRAI	//@checker.0
25	gens	http://Cec.ecore/1.0	Personne	uniqueNameOne_P1_VN	gens::Personne.allInstances	invariant	VRAI	//@checker.0
26	gens	http://Cec.ecore/1.0	Personne	uniqueNameOne_P1_VI	gens::Personne.allInstances	invariant	VRAI	//@checker.0
27	gens	http://Cec.ecore/1.0	Personne	uniqueNameOneNot_P0_VI	gens::Personne.allInstances	invariant	VRAI	//@checker.0
28	gens	http://Cec.ecore/1.0	Personne	uniqueNameOneNot_P1_VI	gens::Personne.allInstances	invariant	VRAI	//@checker.0
29	gens	http://Cec.ecore/1.0	Personne	femaleHasNofemme	self.gender.=(gens::Gender	invariant	VRAI	//@checker.0
30	gens	http://Cec.ecore/1.0	Personne	maleHasNoMari	self.gender.=(gens::Gender	invariant	VRAI	//@checker.0
31	gens	http://Cec.ecore/1.0	Personne	abcName_P0_VT	##########	invariant	VRAI	//@checker.0
32	gens	http://Cec.ecore/1.0	Personne	abcName_P0_VN	##########	invariant	VRAI	//@checker.0
33	gens	http://Cec.ecore/1.0	Personne	abcName_P0_VI	##########	invariant	VRAI	//@checker.0
34	gens	http://Cec.ecore/1.0	Personne	abcName_P1_VT	##########	invariant	VRAI	//@checker.0
35	gens	http://Cec.ecore/1.0	Personne	abcName_P1_VN	##########	invariant	VRAI	//@checker.0
36	gens	http://Cec.ecore/1.0	Personne	abcName_P1_VI	##########	invariant	VRAI	//@checker.0
37	gens	http://Cec.ecore/1.0	Personne	abcNameDotShortcutP0_VT	##########	invariant	VRAI	//@checker.0
38	gens	http://Cec.ecore/1.0	Personne	abcNameDotShortcutP0_VN	##########	invariant	VRAI	//@checker.0
39	gens	http://Cec.ecore/1.0	Personne	abcNameDotShortcutP0_VI	##########	invariant	VRAI	//@checker.0
40	gens	http://Cec.ecore/1.0	Personne	abcNameDotShortcutP1_VT	##########	invariant	VRAI	//@checker.0
41	gens	http://Cec.ecore/1.0	Personne	abcNameDotShortcutP1_VN	##########	invariant	VRAI	//@checker.0
42	gens	http://Cec.ecore/1.0	Personne	abcNameDotShortcutP1_VI	##########	invariant	VRAI	//@checker.0
43	gens	http://Cec.ecore/1.0	Personne	personsWithUndefinedMari	gens::Personne.allInstances	invariant	VRAI	//@checker.0
44	gens	http://Cec.ecore/1.0	Personne	personsWithUndefinedMari	gens::Personne.allInstances	invariant	VRAI	//@checker.0
45	gens	http://Cec.ecore/1.0	Personne	personsWithUndefinedMari	gens::Personne.allInstances	invariant	VRAI	//@checker.0
46	gens	http://Cec.ecore/1.0	Personne	flatten_P0_VT	##########	invariant	VRAI	//@checker.0
47	gens	http://Cec.ecore/1.0	Personne	flatten_P0_VN	##########	invariant	VRAI	//@checker.0

Fig. 1. Détail Fichier TOPCASED

FICHIER RESULTAT COMPARAISON DE TOPCASED/NEPTUNE



	M	N	O	P	Q	R	S	T	U
1	name3	content	type	result topcase	ev	Résultat Neptu	Neptune corrigé	son Neptune	explication
3	enumGender	self.gender.=(gens::Gender::female).or(self.genc	invariant	VRAI	//@ch	null	null	problème	dans Neptune le genre est parfois = "" dans le modèle Cec.gens, dans TOI renseigné dans l'application mais pas dans le métamodèle
7	uniqueNameForAll1_P0_VT	gens::Personne.allInstances()->forAll(self2 : Per	invariant	VRAI	//@ch	false	FAUX	problème	PROBLEME NEPTUNE
8	uniqueNameForAll1_P0_VN	gens::Personne.allInstances()->forAll(self2 : Per	invariant	VRAI	//@ch	false	FAUX	problème	PROBLEME NEPTUNE
9	uniqueNameForAll1_P1_VT	gens::Personne.allInstances()->forAll(self2 : Per	invariant	VRAI	//@ch	false	FAUX	problème	PROBLEME NEPTUNE
10	uniqueNameForAll1_P1_VN	gens::Personne.allInstances()->forAll(self2 : Per	invariant	VRAI	//@ch	false	FAUX	problème	PROBLEME NEPTUNE
11	uniqueNameForAll2_P0_VT	gens::Personne.allInstances()->forAll(p1 : Persc	invariant	VRAI	//@ch	false	FAUX	problème	PROBLEME NEPTUNE
12	uniqueNameForAll2_P0_VN	gens::Personne.allInstances()->forAll(p1 : Persc	invariant	VRAI	//@ch	false	FAUX	problème	PROBLEME NEPTUNE
13	uniqueNameForAll2_P1_VT	gens::Personne.allInstances()->forAll(p1 : Persc	invariant	VRAI	//@ch	false	FAUX	problème	PROBLEME NEPTUNE
14	uniqueNameForAll2_P1_VN	gens::Personne.allInstances()->forAll(p1 : Persc	invariant	VRAI	//@ch	false	FAUX	problème	PROBLEME NEPTUNE
30	maleHasNoMari	self.gender.=(gens::Gender::male).implies(Set (	invariant	VRAI	//@ch	null	null	problème	PROBLEME NEPTUNE dans le genmodel on ne sait pas qui est marrié à génération du métamodèle TOPCASED qui est incomplet par rapport à cel l'application
46	flatten_P0_VT	##	invariant	VRAI	//@ch	false	FAUX	problème	PROBLEME NEPTUNE dans le genmodel on ne sait pas qui est marrié à
47	flatten_P0_VN	##	invariant	VRAI	//@ch	false	FAUX	problème	PROBLEME NEPTUNE dans le genmodel on ne sait pas qui est marrié à
48	flatten_P0_VI	##	invariant	VRAI	//@ch	false	FAUX	problème	PROBLEME NEPTUNE dans le genmodel on ne sait pas qui est marrié à
49	flatten_P1_VT	##	invariant	VRAI	//@ch	false	FAUX	problème	PROBLEME NEPTUNE dans le genmodel on ne sait pas qui est marrié à
50	flatten_P1_VN	##	invariant	VRAI	//@ch	false	FAUX	problème	PROBLEME NEPTUNE dans le genmodel on ne sait pas qui est marrié à
51	flatten_P1_VI	##	invariant	VRAI	//@ch	false	FAUX	problème	PROBLEME NEPTUNE dans le genmodel on ne sait pas qui est marrié à
53									
54									
55									
56									
57									
58									
59									
60									
61									
62									
63									
64									
65									

Fig. 1. Détail Fichier de la comparaison des résultats NEPTUNE/TOPCASED

LISTE DES FIGURES

LISTE DES TABLEAUX

Etude de la conformité à la norme OCL des logiciels critiques Neptune et Topcased qui utilisent des méta-modèles.

RESUMER

L'étude de la conformité à la norme Ocl des outils Neptune et Topcased a permis de soulever deux problèmes principaux dans la norme OCL : d'une part le traitement des valeurs « null », est-ce que OCL doit fonctionner en logique binaire (algèbre de Boole) comme actuellement ou doit-il évoluer vers la logique ternaire comme le langage SQL voire même quaternaire ou plus avec la logique floue ?
D'autre part le problème de l'indéterminisme d'OCL au travers des requêtes utilisant des relations d'ordre qui ne sont pas précisées comme dans la commande asSequence().
Cette étude a permis de tester les outils Neptune et Topcased au travers de campagnes de test en s'inspirant de la recherche de M.Gogolla [GKB08a] et de corriger Neptune.Quelques problèmes sont apparu concernant Topcased, notamment sur la syntaxe de « iterate » et le typage des valeurs.
Les tests ont été réalisés grâce à l'outil de test Salome TMF qui a été programmé à cet effet.

Mots clés : méta-modèle,OCL, Neptune,Topcased, Salome TMF, logique ternaire,campagne de test, script Beanshell, Ingénierie dirigée vers les modèles,XMI,JAVA

SUMMARY

The study of the conformity with the standard Ocl of tools Neptune and Topcased allowed to raise two main problems in the standard OCL: on one hand the processingof the values " null ", OCL has to work in binary logic (Boolean algebra) as at present or does he have to evolve towards the ternary logic as the language SQL even quaternary or more with the vague logic?
On the other hand the problem of the indéterminism of OCL through the requests using relations of order which are not specified as in the asSequence command order (). This study allowed to test tools Neptune and Topcased through campaigns of test by being inspired by the research of M.Gogolla [GKB08a] and to correct Neptune. Some problems seemed concerning Topcased, in particular on the syntax of " iterate " and the typage of the values. The tests were realized thanks to the tool of test Salome TMF who was processed for that purpose.

Keywords: Méta-model, OCL, Neptune, Topcased, Salome TMF, ternary logic, test campaign, Beanshell script, Model Driven Engineering, XMI, JAVA