Chapitre 3

Le génie logiciel et l'IDM : une approche unificatrice par les modèles

3.1. Introduction

Ce chapitre a pour objectif de mettre en évidence le rôle unificateur de l'ingénierie des modèles (IDM) vis-à-vis des différentes activités du cycle de développement du logiciel, et de manière plus large, pour le génie logiciel.

Pris dans un sens large, le génie logiciel englobe en effet les « règles de l'art » de l'ingénierie de la réalisation des systèmes manipulant de l'information, que l'on appelle encore systèmes à logiciel prépondérant. Pour les systèmes de plus en plus complexes que l'on cherche à construire, il s'agit de trouver un compromis entre un système parfaitement bien conçu (qui pourrait demander un temps infini pour être construit) et un système trop vite fait (qu'il serait difficile de mettre au point et de maintenir), en conciliant trois forces largement antagonistes : les délais, les coûts, et la qualité. Il ne s'agit donc pas de produire le système parfait dans l'absolu, mais bien de produire, en respectant un coût et des délais raisonnables, un système suffisamment bon, compte tenu de son contexte d'utilisation.

Le principal problème du génie logiciel ne se pose plus en termes de « donnez-moi une spécification immuable et je produis une implantation de qualité » mais plutôt en « comment réaliser une implantation de qualité avec des spécifications continuellement mouvantes ». De plus, il arrive de plus en plus fréquemment de ne plus avoir à produire un produit donné à un moment donné, mais simultanément

---

Chapitre rédigé par Jean-Marc JÉZÉQUEL, Sébastien GÉRARD, Chokri MRAIDHA et Benoit BAUDRY.

toute une gamme (ou famille) de produits pour prendre en compte des variations de fonctionnalités ou d'environnements [JOS 98]. A cet égard, on peut citer des constructeurs emblématiques tels que Microsoft, qui adaptent certains de leurs logiciels à des dizaines de plates-formes différentes ainsi qu'à des centaines d'environnements culturels, ou encore l'éditeur de jeux vidéos Gameloft qui a développé rien moins que sept cents versions du jeu *Tom Clancy's Splinter Cell Chaos Theory*, conçu par Ubisoft, afin de l'adapter à quelque cent quatre-vingts modèles de téléphones portables, dans différentes langues et dans soixante-cinq pays (des acteurs du secteur expliquent que chaque nouvelle version peut coûter entre 1 000 et 2 000 dollars). La maîtrise de cette double variabilité spatiale et temporelle est donc devenue un des enjeux majeurs de l'industrie du logiciel.

Quel qu'il soit, un processus de développement logiciel englobe un certain nombre d'activités (comme l'expression des besoins, l'analyse, la conception, l'implantation ou encore la validation) qui chacune produit un ou plusieurs artefacts tels que documentation, diagrammes, codes sources, fichiers de configuration, fiches de tests, rapports de qualification, etc. Ces artefacts donnent de multiples points de vue sur le logiciel en cours de développement et, en pratique, sont souvent indépendants les uns des autres. Un problème ici, est d'être capable d'assurer une cohérence entre ces vues, ou au minimum une tracabilité entre les éléments des différents artefacts.

En réponse à ces nouveaux besoins de développement, et aux nombreuses activités citées ci-dessus, de nombreuses solutions technologiques ont été proposées. Ces solutions sont soit très générales (XML, langages orientés objet, UML, AOP, etc.), soit très spécifiques et dédiées à un domaine d'application (systèmes d'exploitation temps-réel, profils UML, langages spécifiques de domaines, etc.). Cependant, aucune de ces solutions n'est parfaitement adaptée à toutes les activités du génie logiciel : au cours du développement, il faudra souvent choisir plusieurs solutions en fonction de l'étape (cahier des charges, conception, validation, etc.) que l'on souhaite réaliser.

L'idée que nous souhaitons mettre en avant dans ce chapitre est la suivante : l'ingénierie dirigée par les modèles offre un cadre méthodologique et technologique qui permet d'unifier différentes façons de faire dans un processus homogène. Il est ainsi possible d'utiliser la technologie la mieux adaptée à chacune des étapes du développement, tout en ayant un processus global de développement qui soit unifié dans un paradigme unique.

L'IDM permet cette unification grâce à l'utilisation importante des modèles (qui peuvent être exprimés dans des formalismes différents) et des transformations automatiques (ou interactives) entre les modèles. L'utilisation intensive de modèles permet un développement souple (c'est-à-dire prompt à réagir aux spécifications

mouvantes) et itéractif, grâce aux raffinements et enrichissements par transformations successives. Par ailleurs, les transformations permettent de passer d'un espace technique à un autre (par exemple de UML vers les graphes ou vers du XML). Ainsi, les transformations permettent de choisir l'espace technique et le formalisme le plus adapté à chaque activité (par exemple conception en UML, génération de test sur des graphes, configuration lors du déploiement *via* des fichiers XML), tout en ayant un cadre méthodologique unique, l'IDM.

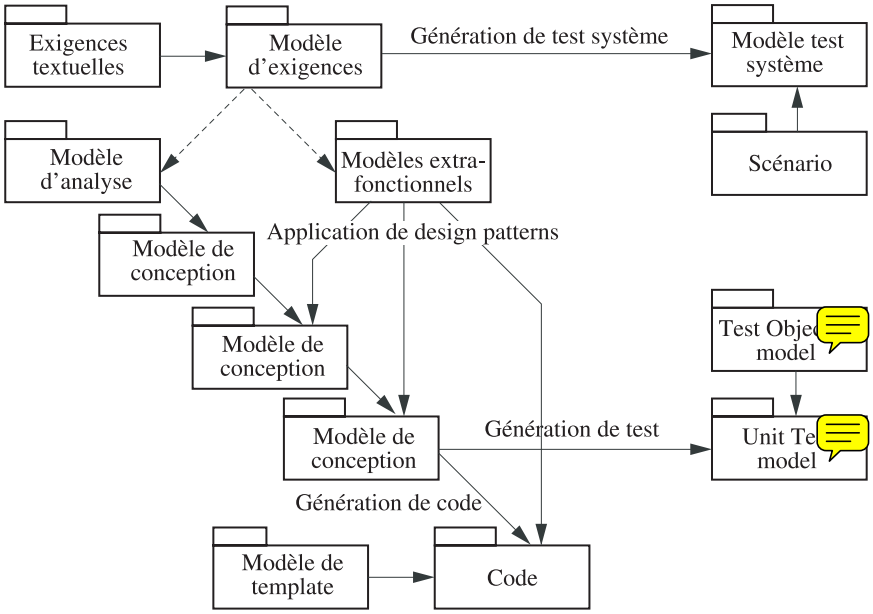


Figure 3.1. *Quelques exemples de transformations de modèles dans un cycle de développement*

L'IDM offre également une solution intéressante pour résoudre le problème de la spécification mouvante en capitalisant une partie du développement grâce aux modèles. En effet, jusqu'à présent, seuls les modules de code source étaient réutilisés, les autres artefacts étant pour la plupart de nature informelle. Or, un des enjeux de l'IDM consiste à formaliser une partie de ces artefacts en tant que modèles réutilisables. Par exemple, un même modèle d'analyse pourra être réutilisé et raffiné vers différents modèles de conception (par exemple, le modèle d'un système offrant certaines fonctionnalités peut être raffiné en prenant en compte différentes contraintes de temps ou d'espace). La formalisation des artefacts en tant que modèles permet également de définir des règles qui garantissent la cohérence entre

les différents artefacts afin d'obtenir un modèle global homogène. Par exemple, on pourra formaliser la relation entre les modèles statiques et dynamiques d'une application.

L'IDM permet aussi de capitaliser le savoir-faire de modélisation grâce aux transformations de modèles. Actuellement, le savoir-faire qui permet la création des différents artefacts au cours du développement reste essentiellement empirique et localisé dans la tête des développeurs. Or, il apparaît qu'une grande partie de la complexité réside au moins autant, si ce n'est plus, dans la *manière* de concevoir un système que dans l'élaboration de ses fonctionnalités. Une idée forte de l'IDM est de pouvoir mécaniser la production d'un modèle donné à partir d'une part, d'un certain nombre de modèles d'entrée, et d'autre part d'un savoir-faire (par exemple de conception) réifié sous forme d'une transformation de modèles [INR 05]. La figure 3.1 illustre plusieurs exemples de transformations de modèles.

La suite de ce chapitre illustre l'intérêt de l'IDM à travers plusieurs exemples pris à différents niveaux du cycle de développement du logiciel : élaboration d'exigences dont la cohérence est controlée automatiquement ; application automatique de patrons de conception ; génération de code ; et finalement génération de tests.

3.2. L'ingénierie des exigences

L'ingénierie des exigences regroupe plusieurs activités pour produire un modèle complet et évolutif : définir les exigences, les modéliser et les analyser, les valider, les faire évoluer et les documenter. Ces activités sont clairement définies dans [NUS 00].

Dans cette section, nous illustrons une approche particulière de l'ingénierie des exigences qui bénéficie de techniques basées sur les modèles pour l'activité de modélisation et d'analyse. De plus, cette approche pour la modélisation facilite l'évolution (grâce à la traçabilité) et la communication (transformation vers du texte, ou un autre formalisme mieux adapté à la communication).

3.2.1. *Vision globale*

La figure 3.2 illustre une succession de transformations de modèles (numérotées de 1 à 4 sur la figure), conformes à des métamodèles différents, permettant de produire un modèle d'exigences simulable. Ce processus est un exemple d'approche dirigée par les modèles pour formaliser les exigences et aider le concepteur des exigences dans les phases de définition et de modélisation des exigences. Cette série

de transformations prend en entrée, une description des exigences exprimées dans un langage naturel contraint, et produit un modèle d'exigences à base de *Use Cases* dans lequel chaque cas est défini par une pré et une postcondition.

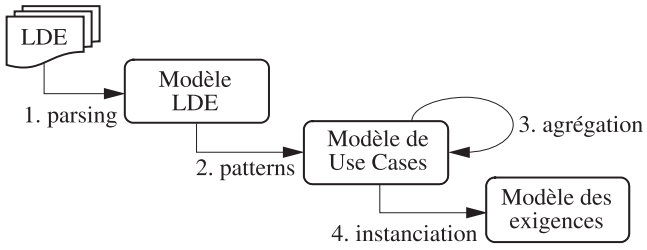


Figure 3.2. *Modélisation des exigences par transformations de modèles*

Un exemple de langage pour la modélisation des exigences est le LDE (langage de description des exigences [NEB 04]). Même s'il restreint l'usage du langage naturel, celui-ci ~~permet toutefois~~ d'écrire des phrases ambigües, incomplètes, voir incorrectes et de ne pas modifier radicalement la pratique de rédaction des exigences. L'expression des exigences en LDE permet :

– d'unifier un vaste ensemble d'exigences vers un plus petit ensemble de cas d'utilisation représentant la synthèse des services proposés par le système ;

– de rejeter les exigences ambigües ou incomplètes ;

– et d'identifier les acteurs du système et les exigences qui ne sont pas du bon niveau d'abstraction.

3.2.2. *Des exigences textuelles au modèle des exigences*

L'analyse des exigences textuelles consiste à analyser le texte LDE pour produire un modèle d'exigences qui soit ~~mainpulable~~ automatiquement par une machine (pour la simulation, la génération de test, la vérification de propriétés, etc.). Cette analyse consiste à appliquer les quatre transformations de modèles décrites figure 3.2.

La première transformation de la figure 3.2 analyse les exigences textuelles d'un système, exprimées en LDE et produit un modèle de ce texte qui peut être analysé automatiquement. Si, au cours de cette transformation, une exigence ne peut pas être analysée syntaxiquement, elle doit être modifiée jusqu'à ce qu'elle soit correcte vis-à-vis de la syntaxe LDE.

La seconde transformation analyse le modèle LDE. Cette analyse consiste à repérer des motifs dans le modèle pour extraire des cas d'utilisation. Cette transformation permet de repérer et d'éliminer les exigences ambigües ou sans « signification ». Le résultat de cette seconde transformation est un modèle modèle fonctionnel sous forme de cas d'utilisation contractualisés, c'est-à-dire augmentés de pré et post conditions.

Dans la figure 3.2, on remarque que la troisième transformation du processus se fait sur les cas d'utilisation, générés au cours de l'interprétation du modèle LDE. Cette transformation prend un ensemble de cas d'utilisation et agrège les pré et post conditions concernant le même cas. Elle produit un nouvel ensemble de cas d'utilisation dans lequel chaque cas n'est présent qu'une fois avec ses contrats agrégés complets.

L'interprétation peut aussi générer une ébauche de modèle statique ~~est~~ une synthèse des différentes composantes du système, telles qu'elles peuvent être déduites des exigences. Pour rester compatible avec le formalisme UML, cette ébauche peut être présentée sous la forme ~~de~~ diagramme de classe d'analyse stéréotypé. L'ébauche n'est pas à proprement parler un modèle de classes d'analyse, mais c'est un support à l'élaboration d'un tel modèle.

Enfin, la dernière transformation produit un modèle instancié des exigences. Cette transformation génère un graphe des comportements possibles des exigences à partir du modèle de cas d'utilisation et d'une configuration initiale (un état initial du système). Il est alors possible de simuler le modèle des exigences. Pratiquée exhaustivement, la simulation permet tout d'abord de détecter les exigences totalement déconnectées des autres exigences. De telles exigences doivent être étudiées pour s'assurer qu'elles ont bien lieu d'être dans le système décrit, s'il n'y a pas lieu de scinder le système en sous-systèmes plus ou moins indépendants, ou si des liens existant entre les exigences n'ont pas été omis ou sous-spécifiés.

La simulation est également et surtout un moyen de valider les exigences, ou d'y détecter des erreurs ou des manques. Ainsi, la simulation permet d'améliorer et de compléter les exigences.

3.2.3. *Synthèse*

Le processus présenté est une illustration des bénéfices possibles de l'IDM pour l'ingénierie des exigences. Il permet, à partir des exigences exprimées dans un langage naturel contraint, d'appliquer une série de transformations de modèles pour obtenir un modèle fonctionnel simulable. Notons qu'il est possible d'ajouter aux

différentes transformations des fonctions de traces qui permettent de tracer les liens entre le modèle des exigences produit et les exigences initiales.

3.3. Application de patrons de conception (*design patterns*)

Un modèle d'analyse a pour objectif de modéliser le domaine du problème de façon à ce qu'il puisse être parfaitement compris et servir de base stable à la conception. Sur cette base, l'activité de conception de logiciels a pour objectif d'aller vers un modèle de conception détaillé qui sera directement implantable, en prenant en compte les forces non fonctionnelles implicites (telles que fiabilité, sécurité, performance, ponctualité, consommation d'énergie, qualifié de service, flexibilité, évolutivité, maintenabilité, etc.). La généralisation dans l'industrie d'une approche par objets de la conception de logiciel a permis de focaliser cette tâche de conception sur la description des schémas (ou motifs) d'interactions entre objets, et d'assigner des responsabilités à des objets individuels de telle sorte que le système résultant de leur composition ait les propriétés extra-fonctionnelles voulues. L'aspect récurrent de certains de ces schémas d'interactions entre objets a conduit à vouloir les cataloguer. On les appelle alors *design patterns* [GAM 95] notion que l'on peut traduire par patrons de conception, pour mettre en évidence le parallèle avec la notion de forme prédéfinie existant dans un patron en couture qui laisse en pratique de nombreux degrés de liberté dans son utilisation.

Le processus de conception peut alors être vu comme une transformation progressive du modèle d'analyse en un modèle d'implantation par l'application successive d'un ensemble de règles de conceptions. Idéalement, ces règles de conception devraient toutes pouvoir être exprimées comme des patrons de conception. On parle alors de « langage de patrons » (*pattern language* en anglais), dans le sens où chaque patron de conception individuel joue le rôle d'un mot dans un vocabulaire, et où leur articulation cohérente forme des phrases faisant sens et permettant de résoudre des problèmes globaux de conception [GAM 95].

Au cours de la décennie 1995-2005, on est ainsi passé progressivement d'une approche artisanale et approximative de la conception à une application rationnelle de solutions ayant fait l'objet d'un catalogage systématique parce qu' issues des meilleures pratiques du domaine. Aujourd'hui, on aborde un nouveau défi concernant l'automatisation de l'application de ces solutions de conception. La difficulté est que les solutions de conception proposées par un *design pattern* particulier diffèrent presque toujours dans leurs détails tout en restant semblables dans leurs principes. De plus, un patron de conception n'est par principe jamais « complet », dans la mesure où il doit permettre une infinité de variation autour d'un même thème. Cependant, si la reconnaissance des problèmes de conception et le choix d'une variante particulière de l'application d'un *design pattern* restent des

activités difficilement mécanisables (sauf techniques d'intelligence artificielle), il n'en est pas de même pour la phase d'application de la solution de conception.

Depuis [LEG 00], on dispose en effet d'un formalisme de description des *design patterns* au niveau du métamodèle d'UML, suffisamment précis pour être exploitable par un outil d'application automatique. Par transformation de modèle, il est ainsi possible de passer automatiquement d'un modèle de conception où l'on mentionne qu'un *design pattern* doit être utilisé (la figure 3.3 illustre un exemple pour le pattern *Command*) à un modèle de conception montrant le résultat de l'application d'une variante particulière de ce pattern.

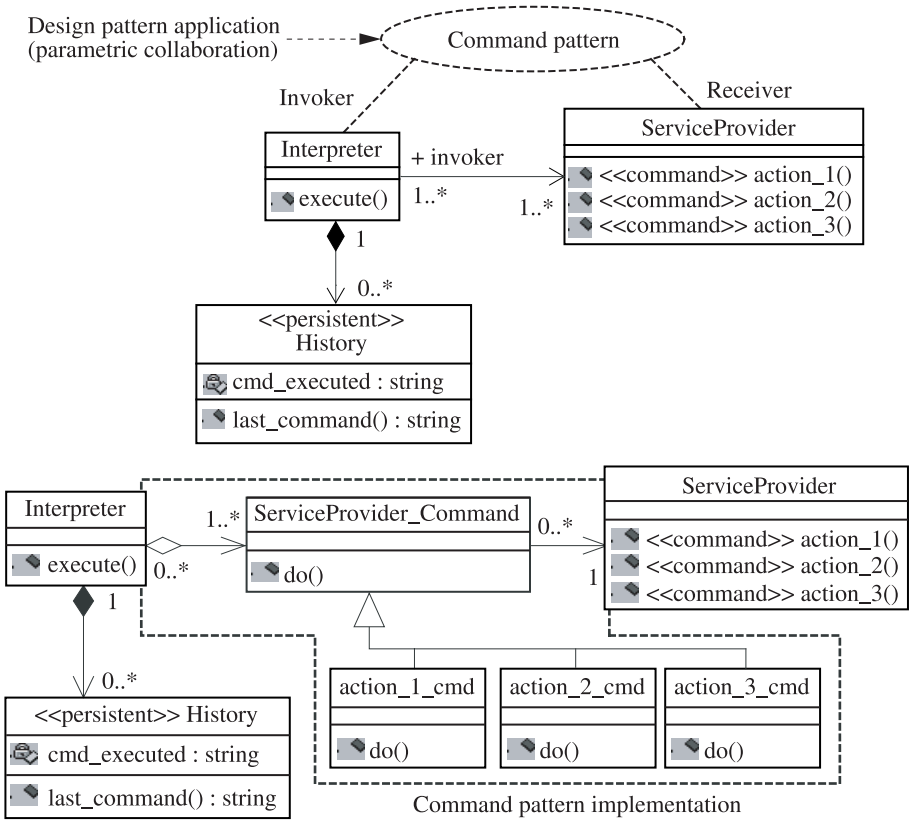


Figure 3.3. *Exemple d'une implantation du patte mmand*
(Traduction des termes de cette figure ?)

La difficulté est alors de fournir de telles transformations de modèles de manière à préserver l'extensibilité du jeu de solutions, proposé par un patron de conception donné. Pour cela, deux approches sont habituellement envisagées, l'une fondée sur la programmation logique, qui par sa nature déclarative ne limite pas *a priori* l'ensemble des solutions possibles. L'autre, probablement plus répandue, consiste à utiliser pour la conception de ces transformations de modèles, les mêmes principes de conception par objets qui président à la conception de l'application : principe d'ouverture-fermeture modulaire apporté par l'héritage et la liaison dynamique, et patrons de conceptions permettant l'extensibilité des solutions proposées. Les patrons de conception sont alors utilisés en quelque sorte réflexivement pour réaliser la flexibilité nécessaire à l'implantation des transformations de modèles permettant l'application de patrons de conceptions.

En pratique au niveau industriel, il existe aujourd'hui de nombreux prototypes d'outils qui automatisent plus ou moins cette application de patrons de conception. Il faut bien avouer cependant que la plupart d'entre eux ont encore une certaine marge de progression en ce qui concerne la flexibilité et le paramétrage des solutions de conception proposées aux concepteurs.

3.4. Du modèle de l'application à son exécution

Le principe même de l'IDM est de capitaliser le savoir faire au niveau des modèles et non plus au niveau du code source. On passe des approches centrées code vers les approches centrées modèles tout en ayant comme objectif de générer/synthétiser une application sous la forme d'un code, binaire ou autre, afin de pouvoir l'exécuter ou l'interpréter sur une machine matérielle donnée.

Il y a plusieurs techniques possibles de synthèse d'une application à partir d'un modèle. Chacune de ces techniques possède un nombre plus ou moins important de pré requis et nécessite un nombre plus ou moins important de transformations du modèle d'entrée. Nous allons dans cette section nous attacher à décrire trois techniques de synthèse d'applications logicielles : l'interprétation de modèles ; la compilation de modèles ; et la génération de code.

Quelle que soit la technique mise en œuvre pour passer du modèle d'un système à une application logicielle exécutable, il est nécessaire que la sémantique du modèle à exécuter soit clairement définie et surtout non ambigüe.

De plus la complétude d'un modèle, signifiant ici que « avons-nous toute l'information nécessaire à l'exécution », va influer directement sur le niveau d'exécution atteignable par l'une ou l'autre des techniques.

3.4.1. *L'interprétation de modèles*

Le principe de l'interprétation de modèles repose sur l'existence d'une machine virtuelle qui exécute directement le modèle de l'application spécifiée (par exemple la machine virtuelle Java dans le cas où le modèle est écrit en Java). Dans ce cas, l'interpréteur lit le modèle d'une façon ou d'une autre et calcule un comportement en temps-réel.

Dans l'interprétation de modèles, la portabilité de l'application vers différentes plates-formes d'exécution a l'avantage d'être assurée par le portage de la machine virtuelle vers ces plates-formes. Une fois modélisée l'application peut alors être interprétée sur toutes les plates-formes sur lesquelles la machine virtuelle est implémentée.

Cependant, le portage d'une machine virtuelle vers une plate-forme a un coût et peut donc ne pas être envisageable pour toutes les catégories de systèmes. Un autre inconvénient de cette technique concerne la performance de l'application. En effet, une partie des ressources de la plate-forme va être allouée à l'exécution de la machine virtuelle, ce qui pour certains systèmes aux ressources limitées, peut avoir un sévère impact de performance. Ces problèmes de performance font de l'interprétation de modèles une technique bien adaptée au prototypage rapide d'application ou à la mise au point de systèmes.

3.4.2. *La compilation de modèles*

La compilation de modèles est un processus en une étape qui consiste à transformer le modèle en binaire exécutable sur une plate-forme donnée. Ce principe repose bien entendu sur l'existence d'un compilateur du langage de modélisation pour cette plateforme.

Chaque plate-forme d'exécution possède son propre compilateur de modèles. Cela permet une transformation du modèle en binaire optimisé pour cette plateforme. Le portage d'un modèle d'une plate-forme à une autre repose sur l'existence d'un compilateur pour cette nouvelle plateforme. Le développement de compilateur de modèles doit tenir compte de l'évolution des langages de modélisations (dont les standards sont encore en constante évolution) ainsi que de l'évolution rapide des plates-formes matérielles. La constante évolution des standards de langages de modélisation ainsi que le nombre croissant de plates-formes matérielles font que le développement de compilateurs de modèles n'est pas actuellement chose courante. Il s'agit essentiellement d'un problème économique dû au cout de développement d'un compilateur.

3.4.3. *La génération de code*

La synthèse d'une application à partir de son modèle par génération de code est un processus en deux étapes. La première étape est une transformation du modèle en langage de programmation de haut niveau (C, C++, Java, etc.).

La seconde étape consiste en la compilation de ce programme en binaires exécutables qui font l'application par utilisation d'un compilateur support du langage ciblé lors de la première étape.

Les approches par génération de code sont souvent qualifiées d'approches par traduction, où des gabarits (*template*) décrivent la traduction des éléments du langage de modélisation en langage de programmation de haut niveau [MEL 02]. Pour la génération de code, on peut considérer deux approches. Une première qui consiste à découper cette étape en deux sous étapes : une transformation de modèle à modèle et une transformation de modèle à code source cible (*pretty printing*). La seconde approche se fait en une seule étape ne passant pas par un modèle intermédiaire du langage de programmation cible.

La portabilité des modèles vers différentes plates-formes est accrue par rapport à la technique de compilation de modèles. En effet, la synthèse du binaire de l'application finale repose sur le compilateur existant du langage de programmation généré pour la plate-forme en question. Dans ce cas les évolutions du langage de modélisation se répercutent uniquement sur la maintenance des règles de transformation de modèles en langage de programmation de haut niveau et non à la réécriture d'un compilateur de modèles comme dans le cas précédent. Le modèle, voir le code généré, peut être réutilisé sur une multitude de plates-formes.

La génération de code (les règles de transformation de modèle) peut également être optimisée pour une plate-forme donnée. En effet, l'application d'un patron donné ou le paramétrage d'une règle de transformation peuvent servir à cet effet.
La génération de code est la technique de synthèse d'application la plus répandue, et ce à juste titre car elle semble être la plus adaptée aux besoins actuels de l'ingénierie dirigée par les modèles :

– flexibilité face à l'évolution constante des standards des langages de modélisation. Une réécriture des règles de transformation de modèles pour la génération de code suffit ;

– flexibilité face à la diversité des plates-formes et des systèmes. La génération de code peut être optimisée afin de produire des applications performantes.

Cependant, un problème important reste souvent mal géré quand on utlise cette technique : la mise au point d'application. En effet, le déverminage de modèles exécutés par génération de code s'avère souvent une tâche difficile et souvent faite au niveau du code source généré et non au niveau des modèles. Sur ce point, les techniques d'interprétation sont souvent bien plus efficaces.

Il parait donc interessant, compte tenu des technologies actuelles, de s'appuyer sur un interpréteur lors de la phase de mise au point du modèle d'une application, et sur des compilateurs pour la phase de production finale.

3.5. Génération de tests

Le test consiste à détecter des erreurs dans un système, ce qui se traduit par une série d'activités dont le but est d'observer un comportement du système différent de celui attendu. Dans la plupart des cas, le comportement attendu est décrit dans un document de spécification et le système sous test est une implantation qui doit être conforme à cette spécification. Dans le cas de l'ingénierie dirigée par les modèles, la spécification et le système à tester peuvent être tous les deux des modèles. On distinguera alors la spécification comme étant le modèle le plus abstrait des deux modèles.

Les activités pour le test sont nombreuses et font toutes l'objet d'importants travaux pour les mettre au point et les rendre efficaces. Les activités principales sont la génération de cas de test, l'exécution de ces tests sur le système, la production d'un verdict et éventuellement le diagnostic (qui n'est pas toujours considéré comme une activité de test). Dans la suite nous nous intéressons surtout à la génération de cas de test qui est l'activité qui bénéficie le plus clairement de l'IDM.

La génération de test consiste à repérer, dans la spécification, quelles sont les valeurs intéressantes pour tester le système. Pour une génération efficace de cas de test, le modèle de spécification doit être complet et non ambigü du comportement attendu du système. Si ce modèle est disponible, il est possible de définir des critères de couverture de ce modèle pour générer un ensemble de taille raisonnable de cas de test.

3.5.1. Model-based testing

Une technique largement étudiée pour la génération de test est appelée *model-based testing*. Cette technique utilise un modèle de spécification décrit le comportement attendu d'une implantation du système. Un cas de test est alors un

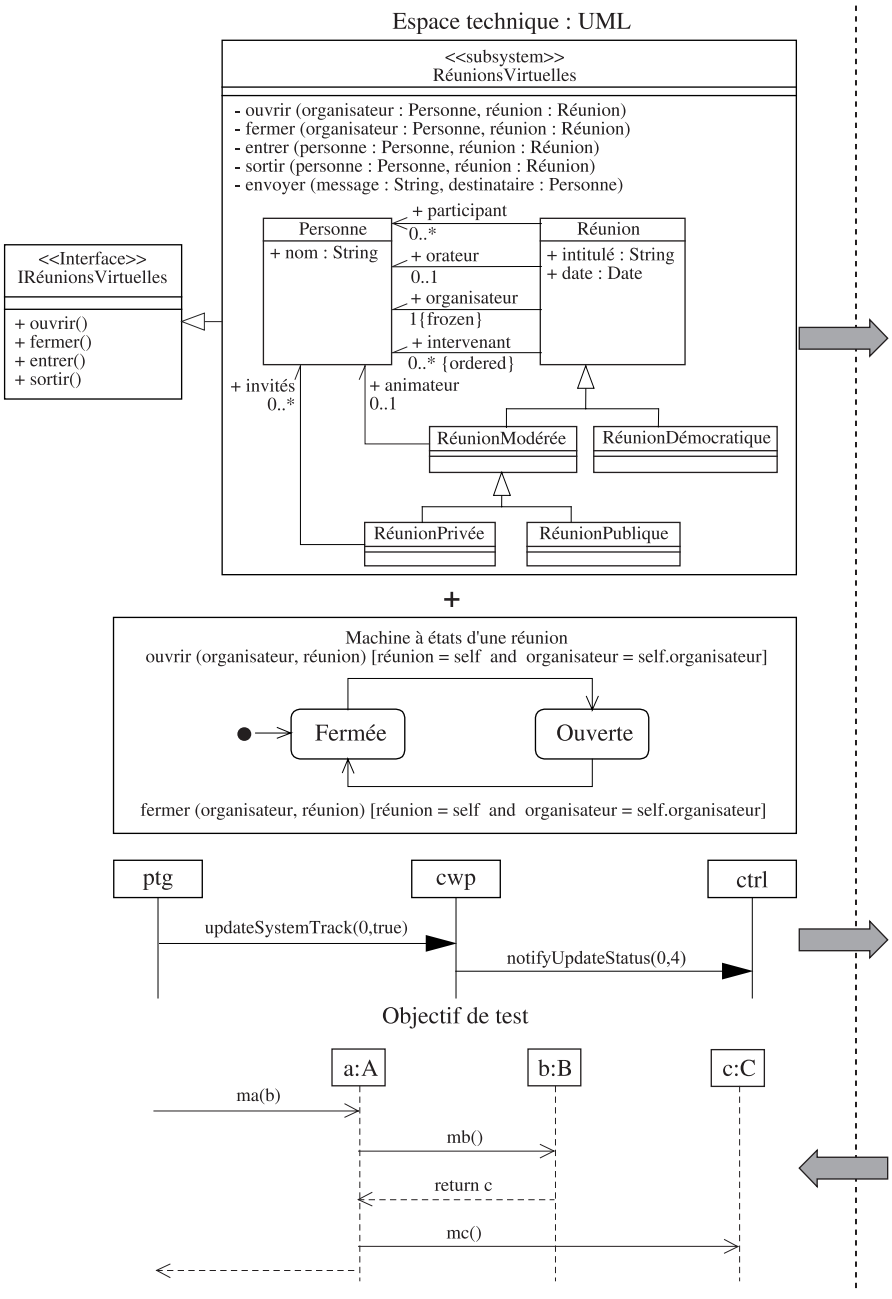
chemin particulier dans l'ensemble des chemins possibles décrits par le modèle de spécification.

L'IDM apporte une solution intéressante pour l'application de cette technique puisqu'elle permet de passer d'un espace technique à un autre par transformation de modèles. En effet, les modèles utilisés pour la génération de test ne sont pas forcément ceux utilisés pour modéliser l'application. Par exemple, UML est largement utilisé au cours du développement d'un logiciel, alors que les techniques de model-based testing s'appuient sur des modèles formels plus adaptés (automate, B, etc.) [JAR 05, LUG 02]. Les transformations de modèles permettent donc d'appliquer ces technologies pour des applications modélisées en UML.

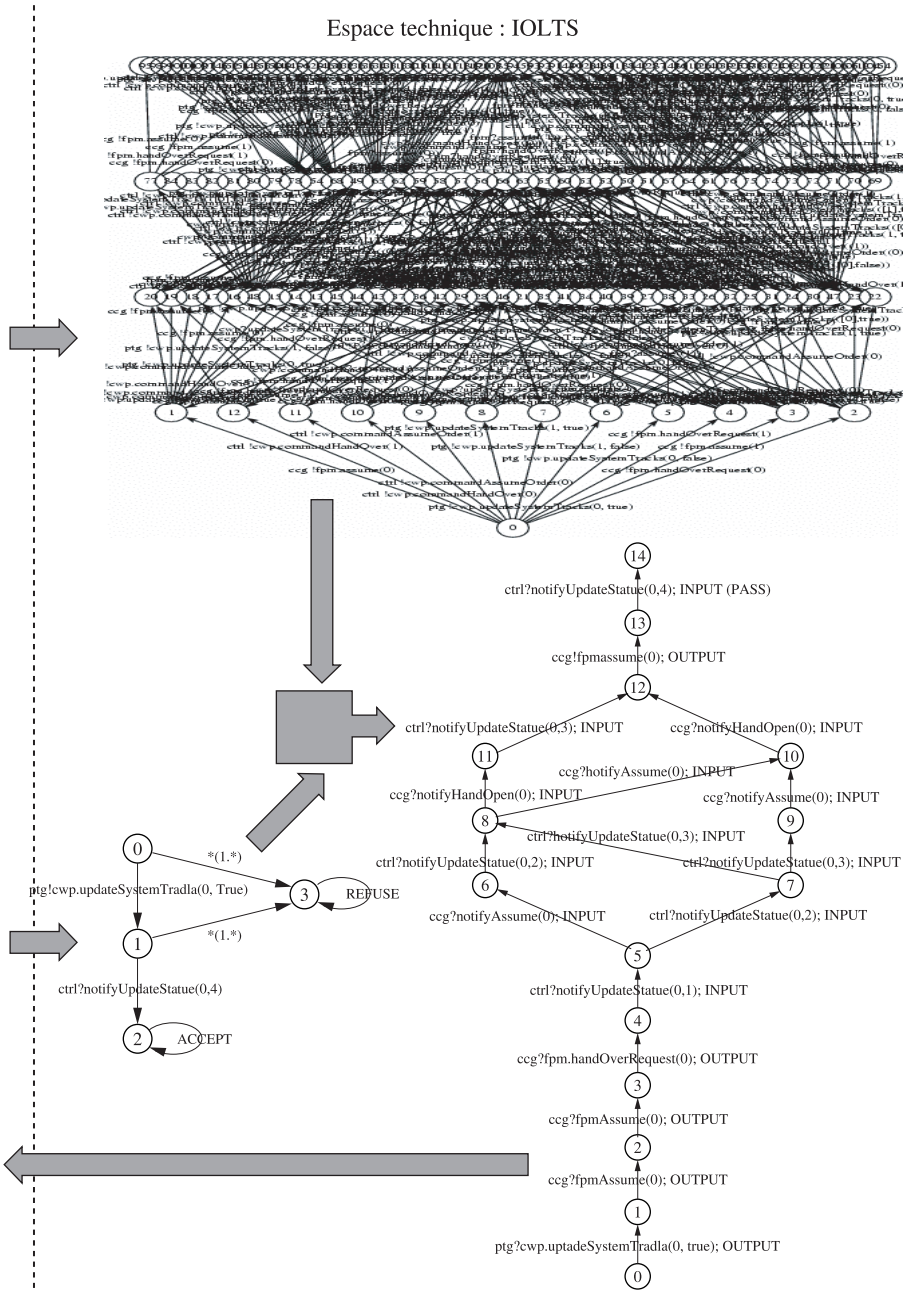
Les modèles UML considérés pour la génération de tests varient légèrement selon les technologies. Les éléments principaux nécessaires dans tous les cas sont la structure du système (diagramme de classes ou de composants) et son comportement (diagramme d'états pour chaque classe). Il est parfois aussi nécessaire de fournir un état initial du système sous la forme d'un diagramme d'objets ou des propriétés attendues sous forme de pré et post conditions ou d'invariants exprimés en OCL. Les modèles UML peuvent ensuite être transformés vers un graphe étiqueté, un réseau de Petri ou un modèle B. Il est alors possible de générer un ensemble de cas de test qui correspondent à un couple d'entrée/sortie, c'est-à-dire une donnée en entrée et une trace d'exécution attendue. Cette génération est guidée par un critère de test (couverture, mutation, etc.) ou par un ensemble d'objectifs de test correspondant à un comportement que l'on veut voir exhiber par le modèle. Une autre transformation de modèles permet d'exprimer les cas de test générés avec UML, par exemple en exprimant les traces d'exécution sous forme de diagrammes de séquence.

La figure 3.4 illustre une approche de génération de tests qui utilise les ponts entre espaces techniques pour la génération de tests. L'application est modélisée avec UML (ici sous la forme de diagrammes de classes et machines à états). Une transformation vers l'espace technique des systèmes de transitions étiquetées (IOLTS) permet d'obtenir un modèle de tous les comportements du système. La technique de génération de test utilise ensuite un objectif de test pour sélectionner un chemin dans le graphe qui correspond à un cas de test.

Cet objectif peut aussi être exprimé sous la forme d'un UML et traduit vers un modèle IOLTS par une transformation de modèle. L'outil de génération de test peut alors synthétiser un cas de test qui pourra ensuite être transformé vers l'espace UML sous forme d'un diagramme de séquence.


Espace technique : UML
<<subsystem>>
RéunionsVirtuelles
- ouvrir (organisateur : Personne, réunion : Réunion)
- fermer (organisateur : Personne, réunion : Réunion)
- entrer (personne : Personne, réunion : Réunion)
- sortir (personne : Personne, réunion : Réunion)
- envoyer (message : String, destinataire : Personne)
<<Interface>>
IRéunionsVirtuelles
+ ouvrir()
+ fermer()
+ entrer()
+ sortir()
Personne
+ nom : String
Réunion
+ intitulé : String
+ date : Date
+ participant
0..*
+ orateur
0..1
+ organisateur
1{frozen}
+ intervenant
0..* {ordered}
+ invités
0..*
+ animateur
0..1
RéunionModérée
RéunionDémocratique
RéunionPrivée
RéunionPublique
+
Machine à états d'une réunion
ouvrir (organisateur, réunion) [réunion = self and organisateur = self.organisateur]
Fermée
Ouverte
fermer (organisateur, réunion) [réunion = self and organisateur = self.organisateur]
ptg
cwp
ctrl
updateSystemTrack(0,true)
notifyUpdateStatus(0,4)
Objectif de test
a:A
b:B
c:C
ma(b)
mb()
return c
mc()

a)



b)

Figure 3.4. *Transformation inter espaces techniques pour la génération de tests à partir de modèles*

3.5.2. *Test structurel et test à partir des exigences*

Outre qu'elle permet d'unifier les techniques de *model-based testing* et les techniques de modélisation basée sur UML, l'IDM est utile à d'autres activités du test.

L'IDM permet de considérer tout artefact de développement comme un modèle, et permet donc de manipuler un programme comme un modèle. C'est un point important pour la génération de test. Cette approche permet de réutiliser les outils de l'IDM (qui manipulent des modèles) pour évaluer des mesures sur un programme ou pour perturber le programme (pour l'analyse de mutation par exemple). Ces outils peuvent alors servir à évaluer des critères de test comme le taux de couverture de code ou le score de mutation d'un ensemble de cas de test.

Plus tôt dans ce chapitre, nous avons présenté de quelle manière l'IDM permet de produire un modèle des exigences à partir de spécifications informelles tout en maintenant un lien de traçabilité entre ces deux vues. Cette approche est aussi importante pour la génération de tests systèmes. Le modèle formel des exigences décrit le comportement attendu du système, et ce modèle comportemental peut être utilisé pour la génération de tests système qui correspondent à des scénarios de haut niveau que l'on voudra évaluer sur l'implantation lorsqu'elle sera disponible. De plus, comme le modèle des exigences est étroitement lié aux exigences informelles, la génération de tests à partir de ce modèle permet d'évaluer précisément la couverture des exigences par les cas de test.

3.6. Conclusion

Au cours de ce chapitre nous avons illustré à travers plusieurs exemples l'apport de l'ingénierie dirigée modèles pour le développement de logiciels. Loin de dresser une liste exhaustive des applications possibles de l'IDM, nous avons montré précisément l'intérêt de de cette approche dans quelques domaines particuliers. Ces exemples ont permis d'illustrer les points forts de l'IDM pour le génie logiciel : l'utilisation intensive, la réutilisation et la transformation de modèles.

3.7. Bibliographie

[GAM 95] GAMMA E., HELM R., JOHNSON R., VLISSIDES J., *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, Reading (MA), 1995.

[INR 05] INRIA, Model transformation at INRIA, http://modelware.inria.fr, 2005.

[JAR 05] JARD C., JÉRON T., « TGV: theory, principles and algorithms. A tool for the automatic synthesis of conformance test cases for non-deterministic reactive systems », *International Journal on Software Tools for Technology Transfer*, **7**(4) : 297-315, 2005.

[JOS 98] BOSCH J., « Product-Line Architectures in Industry: A Case Study », *Proceedings of ICSE'98 (Int. Conference in Software Engineering)*, november 1998.

[LEG 00] LE GUENNEC A., SUNYÉ G., JÉZÉQUEL J.-M., « Precise Modeling of Design Patterns », *Proceedings of UML'00*, p. 482-496. Springer-Verlag, octobre 2000.

[LUG 02] LUGATO D., BIGOT C., VALOT Y., « Validation and automaitc test generation on UML models: the AGATHA approach », *Electronic Notes in Theoretical Computer Science*, **66**(2), 2002.

[MEL 02] MELLOR S.J., BALCER M.J., *Executable UML: a foundation for model-driven architecture*. Addison-Wesley, Boston (MA), 2002.

[NEB 04] NEBUT C., Génération automatique de tests à partir des exigences et application aux lignes de produits logicielles, Thèse de doctorat, 2004.

[NUS 00] NUSEIBEH B., EASTERBROOK S., « Requirements Engineering: A Roadmap », *Proceedings of The Future of Software Engineering (joint event of ICSE'00)*, Limerick, Irelande, p. 35-46, juin 2000.