Méthodes formelles avec UML

Modélisation, validation et génération de tests[1]

Alain Le Guennec

IRISA / Université de Rennes 1
Campus de Beaulieu
F-35042 Rennes Cedex
Tel : +33 2 99 84 25 41
Fax : +33 2 99 84 25 32
Email : Alain.Le_Guennec@irisa.fr

RÉSUMÉ. *Construire et maintenir des systèmes logiciels ou matériels complexes reste une tâche ardue, en particulier dans le domaine des systèmes de télécommunication, où la dimension répartie est omniprésente. L'Unified Modeling Language (UML) s'impose progressivement comme standard de fait lorsqu'il s'agit de modéliser de tels systèmes en suivant une démarche orientée objets. Mais, en dépit de sa grande richesse, ce langage possède une sémantique qui reste informelle et mal définie, ce qui rend difficile l'utilisation directe de méthodes formelles. Nous proposons d'extraire une sémantique opérationnelle à partir de certaines constructions de UML. Nous avons développé un simulateur UML capable de construire le système de transitions étiquetées correspondant à une spécification UML. Ce simulateur peut être couplé à toute une gamme d'outils exploitant ce formalisme, offrant ainsi la possibilité de vérifier certaines propriétés des spécifications UML, ou encore de générer des tests.*

ABSTRACT. *Building and maintaining software or hardware systems remains a difficult task, particularly in the domain of telecommunication systems, which are inherently concurrent. The Unified Modeling Language (UML) is becoming a de facto standard for modeling such systems when following an object-oriented approach. But in spite of its richness, the semantics of this language is still informal and loosely defined. We propose to use some of the constructions offered by UML to extract an operational semantics. We have developed a UML simulator that builds the labeled transition system corresponding to a UML specification. This simulator can be combined with a variety of tools based on this formalism, making it possible to verify certain properties of UML specifications, or to generate test cases.*

MOTS-CLÉS : *UML, Méthodes formelles, Validation, Génération de tests*

KEYWORDS: *UML, Formal methods, Validation, Test generation*

1. Les travaux présentés dans cet article sont partiellement financés dans le cadre du projet RNRT OURAL (convention n^o 992930221).

1. Introduction

Depuis sa standardisation par l'Object Management Group (OMG) en 1997, l'Unified Modeling Language (UML [RTF 99]) s'impose progressivement comme un standard de fait pour la modélisation à objets de systèmes, qu'ils soient logiciels, matériels ou organisationnels. La notation UML est suffisamment complète pour pouvoir remplacer ou compléter les notations à objets l'ayant précédée, et sa souplesse permet de l'utiliser pour modéliser toutes les facettes d'un système, depuis la phase d'identification des besoins jusqu'au test et au déploiement final du système opérationnel.

Bien sûr, plus un système est complexe et plus la nécessité de le valider et de le tester se fait pressante. Tel est le cas notamment des systèmes répartis, dans lesquels la concurrence et l'asynchronisme rendent la compréhension du système particulièrement ardue. L'utilisation de méthodes formelles est un moyen efficace d'améliorer la fiabilité et la qualité des systèmes concurrents. Le secteur des télécommunications est particulièrement en pointe dans ce domaine, grâce aux diverses techniques de descriptions formelles ayant été développées pour faire face à ces exigences de qualité. Ces techniques sont basés sur des notations telles que Estelle, Promela, SDL [CCI 87] ou LOTOS [ISO 85], ayant une sémantique précise et pour lesquelles l'utilisateur dispose d'outils permettant d'aborder la vérification automatique.

Nous proposons d'adapter ces techniques et outils à UML afin que les développements s'appuyant sur cette notation puissent en bénéficier. Il ne s'agit pas pour nous de traduire une spécification UML dans un autre langage formel afin d'utiliser directement les outils qui lui sont dédiés. En effet, bien que possible dans une certaine mesure comme le montrent les travaux mentionnés en section 3.2, une telle conversion se heurte parfois aux spécificités propres au formalisme cible. Du fait de la richesse d'UML, cette conversion ne pourrait probablement pas se faire sans contorsion de la sémantique que nous souhaitons donner à UML. Par conséquent, nous donnons à UML une sémantique opérationnelle qui lui est propre, au travers du formalisme des *systèmes de transitions étiquetées* (Labelled Transition System ou LTS). Un prototype permettant de produire le LTS d'une spécification UML à la volée a été développé pour démontrer la faisabilité de notre approche.

Le reste de cet article est organisé de la manière suivante : la section 2 présente la notation UML au travers d'un exemple de spécification. Puis nous expliquons dans la section 3 comment il est possible de donner une base formelle à une spécification UML grâce aux LTS. Nous décrivons ensuite le simulateur UML permettant d'obtenir les systèmes de transitions. Ce simulateur peut être couplé à des outils ouverts capables d'exploiter ces systèmes de transitions. La section 4 présente quelques techniques formelles que ce couplage nous a permis d'appliquer à la spécification UML de la section 2, comme la vérification ou la génération de tests.

2. UML dans le cycle de développement logiciel

2.1. *Modélisation d'un système de contrôle aérien*

Dans cette section, nous allons présenter comment un système se modélise en UML. La notation UML sera introduite progressivement lors de la présentation. Afin d'illustrer concrètement notre propos, nous avons choisi de modéliser un système de contrôle aérien (ATC), qui servira de cas d'étude tout au long de cet article.

Un tel système peut être modélisé selon plusieurs points de vue ou à divers niveaux d'abstraction (analyse, conception), chacun donnant lieu à un *modèle* particulier. Chaque modèle contient une description du *système* proprement dit, ainsi qu'une description de son environnement, notamment les *acteurs* qui interagissent avec le système. Si le système est complexe, il est possible de décomposer hiérarchiquement un modèle en sous-modèles et un système en sous-systèmes. Pour simplifier, nous considérerons que la spécification UML ne contient qu'un système, vu à un seul niveau d'abstraction donné (donc un seul modèle).

2.2. *Spécification extérieure et détaillée d'un (sous-)système*

La description d'un (sous-)système se divise en deux parties complémentaires :

Une partie spécification vis-à-vis de l'extérieur qui décrit ce que le système doit être capable de faire en termes de *cas d'utilisation* et d'*opérations* portant sur le sous-système dans sa globalité. Les concepts manipulés sont représentés par des classes ou des types UML.

Une partie détaillée qui décrit comment les cas d'utilisation et les opérations globales sont réalisés au sein du sous-système. En général, un sous-système est réalisé par un ensemble d'objets coopérants entre eux, et toute opération sur le sous-système se traduit par un ensemble d'opérations implicant les objets qui le composent. La partie détaillée comportera donc aussi des classes (qui raffinent les classes de la partie spécification) et les *machines d'états* associées, ainsi que des *collaborations* et *interactions* explicitant leur comportement.

2.3. *Identification des besoins*

La première étape de la modélisation consiste généralement à déterminer précisément les besoins des utilisateurs du système. Pour ce faire, UML propose les cas d'utilisations, hérités de la méthode OOSE. Chaque utilisation possible du système se traduit par une ellipse etiquetée avec le nom du cas d'utilisation, à laquelle est connecté le ou les acteurs concernés par cette interaction avec le système. Il est possible de raffiner les cas d'utilisation : une action complexe peut être réalisée par une combinaison d'actions plus simples (on dit que le cas d'utilisation complexe «inclut» les cas plus simples). Une action alternative ou exceptionnelle peut aussi avoir lieu au

cours d'une utilisation complexe si certaines conditions sont remplies (on dit alors que les cas alternatifs ou exceptionnels «étendent» l'utilisation nominale).

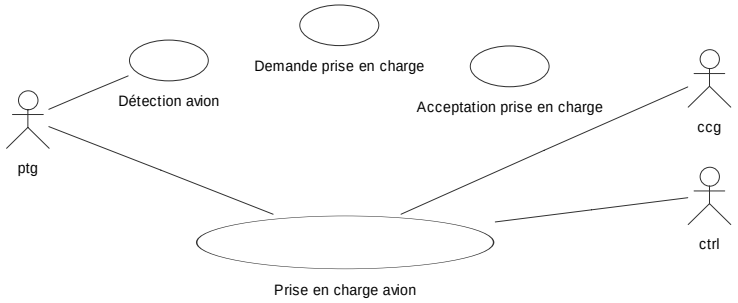


FIG. 1 –. *Cas d'utilisation*

Si l'on souhaite tendre vers plus de rigueur, il est possible de compléter cette vision graphique des besoins par l'utilisation pertinente de pré et post conditions exprimées textuellement. UML fournit à cette fin un langage dédié : l'Object Constraint Language (OCL) [WAR 98]. OCL est un langage ensembliste permettant d'écrire des contraintes sous forme d'expressions pouvant référencer les entités définies dans la spécification UML. Ces contraintes textuelles définissent le *contrat* qui lie le système aux acteurs présents dans son environnement [MEY 92].

2.4. *Le système et son environnement : vue statique*

Les diagrammes de classes de UML présentent la structure statique du système et de son environnement. Y figurent la signature des classes des objets impliqués, les relations d'héritage éventuelles entre classes, ainsi que les associations qui les unissent. Les classes dont le bord est plus épais signalent des objets actifs.

Dans la figure 2, le sous-système représentant l'ATC dans sa globalité a été éclaté, laissant apparaitre la structure interne de celui-ci. La classe `FlightPlan` représente les plans de vol pré-établis des différents avions. La classe `Flight` représente les vols sous la responsabilité de cet ATC, l'attribut `callsign` permettant de faire le lien avec le plan de vol. La classe `FlightPlanManager` représente les gestionnaires de plans de vol. Enfin, la classe `ControllerWorkingPosition` représente la partie du système chargée de coordonner les informations provenant des radars et des controlleurs aériens. L'environnement est modélisé par trois acteurs qui peuvent interagir avec l'ATC en échangeant des messages avec lui : L'acteur `controller` représente le controleur aérien, l'acteur `PTGfacade` représente le système radar fournissant la position des avions dans l'espace aérien de l'ATC, et enfin le `CCGFacade` représente un autre ATC, qui peut déléguer la responsabilité d'un avion lorsque celui change d'espace aérien. Un invariant en OCL assure la cohérence des `callsigns` entre vols et plans de vol :

```
context ControllerWorkingPosition inv:
   flight->forAll(f | flightplan->exists(fp | fp.callsign = f.callsign))
```

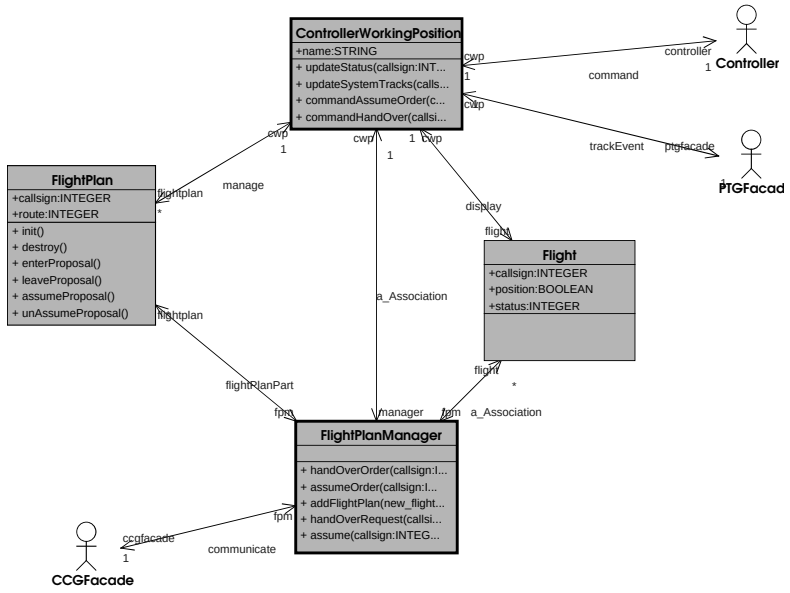


FIG. 2 –. *Vue statique de l'ATC*

2.5. *Modélisation des interactions*

La manière dont les messages sont échangés entre objets pour invoquer les opérations nécessaires à la réalisation d'un cas d'utilisation est une partie délicate à modéliser correctement. Chaque cas d'utilisation met en œuvre une *collaboration* d'objets. Tous les intervenants dans une collaboration possèdent un *rôle* bien précis. Les communications au sein d'une collaboration sont représentées en UML par des interactions, qui définissent un ordre partiel sur les messages échangés entre les différents rôles. Les interactions se représentent graphiquement par des diagrammes de séquences, très similaires aux Message Sequence Charts.

2.6. *Description du comportement réactif du système*

Les interactions décrivent une vision globale, non-localisée, du comportement du système. UML permet de modéliser le comportement selon une vision complémentaire, centrée sur les objets pris individuellement : les machines d'états. La machine d'états attachée à la classe d'un objet décrit le cycle de vie de cet objet, c'est-à-dire la manière dont il réagit lorsqu'un autre objet fait appel à l'une de ses opérations ou lorsque certains événements surgissent. La figure 3 représente ainsi le comportement d'un objet de type «plan de vol», et montre notamment comment son état évolue lorsque ses opérations sont appelées, et les actions exécutées lors des transitions.

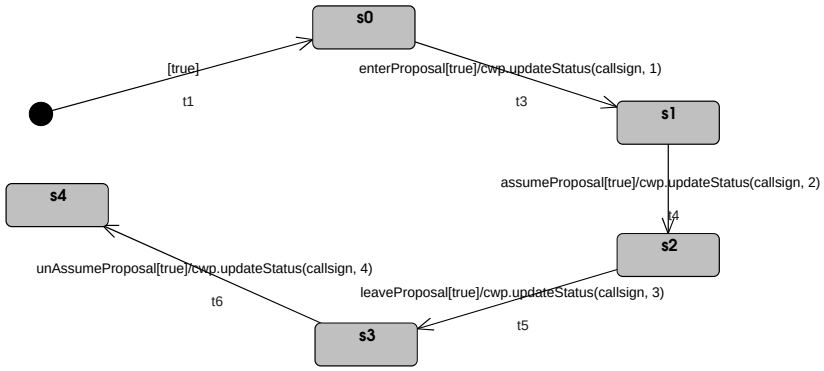


FIG. 3 –. *Machine d'états d'un plan de vol*

Le comportement du système global est donc obtenu par l'exécution conjointe des machines d'états des objets du système et de son environnement, comme nous le verrons plus en détails dans la section 3.2.

2.7. *Configuration du système*

Il nous reste donc à décrire comment le système est configuré en terme d'objets. Nous n'avons en effet décrit jusqu'à présent que le comportement de *classes* d'objets et de collaborations de rôles. Les diagrammes de classes donnent des *configurations potentielles* du système en terme de classes et d'associations. Il n'a pas été fait mention du nombre d'objets de chaque classe, ni de la manière dont ils sont initialement reliés entre eux (c'est-à-dire la topologie du réseau d'objets). Dans le cas où le système est réparti, l'information de localisation des objets est également nécessaire. Toutes ces informations apparaissent sur les diagrammes de UML permettant de représenter des objets. La configuration initiale de l'ATC est ainsi représentée par la figure 4.

Cette configuration permet par exemple de déduire qu'il y a deux plans de vols pré-enregistrés et donc connus du système. Le système est ainsi complètement spécifié et peut à présent être analysé formellement, pour vérifier certaines propriétés (objet de la section 4.1) ou générer des cas de tests (section 4.2). Enfin le système pourra être implanté par des développeurs, qui utiliseront les cas de tests pour s'assurer de la *conformité* de leur implantation vis-à-vis de la spécification.

Il est à noter que plusieurs types de diagrammes UML permettent de représenter une configuration d'objets : Les diagrammes d'objets bien sûr, mais aussi les diagrammes de déploiement (mettant l'accent sur l'aspect «système distribué»), ainsi que les diagrammes de collaborations de niveau instance. Un diagramme de collaboration a cependant vocation à présenter seulement un aspect du système et non sa globalité, en indiquant notamment le rôle joué par chaque objet dans cet aspect du système.

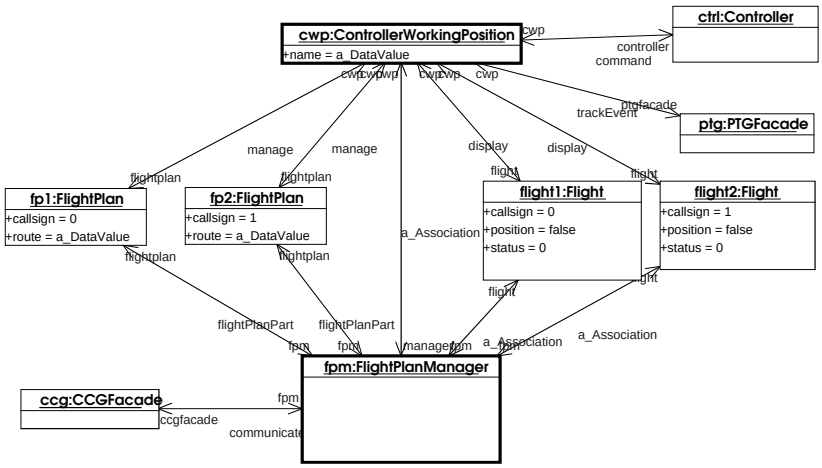


FIG. 4 –. *Configuration initiale de l'ATC*

3. Méthodes formelles avec UML

3.1. *UMLAUT et la boite à outils CADP*

UMLAUT [JÉZ 99] est un outil dédié à la manipulation de modèles UML. UMLAUT a été conçu dès le départ comme un outil ouvert, capable de lire des spécifications UML sauvegardées dans le format standardisé XMI et donc facilement interfaçable avec les ateliers de génie logiciel existants supportant la notation UML. Il peut toutefois être utilisé de manière autonome, via une interface graphique développée en Java. Cette interface supporte les types de diagrammes les plus importants (cas d'utilisations, diagrammes de classes, machines d'états, déploiements, diagrammes de collaborations et bientôt diagrammes de séquences). Les figures présentées précédemment ont d'ailleurs été réalisées avec UMLAUT.

UMLAUT est un outil modulaire et extensible. Outre l'interface graphique et les modules d'importations de modèles, l'outil dispose d'une bibliothèque d'opérateurs de transformations pouvant être appliquées à des modèles UML, ainsi que d'un module permettant d'analyser et d'évaluer des expressions écrites en OCL [WAR 98]. Enfin, UMLAUT donne accès aux techniques formelles grâce à son module de simulation, capable de compiler une spécification UML sous la forme d'un système de transitions étiquetées construit à la volée. Une spécification UML compilée de la sorte est ainsi directement exploitable par les outils de la boite à outils CADP [GAR 98], qui sont tous basés sur la même interface vers un système de transitions (l'API « graph » de CADP) qu'implémente UMLAUT. Parmi les outils disponibles, on peut citer un simulateur interactif permettant d'explorer le comportement du système, un *model checker*, et le générateur de tests TGV [JÉR 99].

3.2. *Vers une sémantique pour UML*

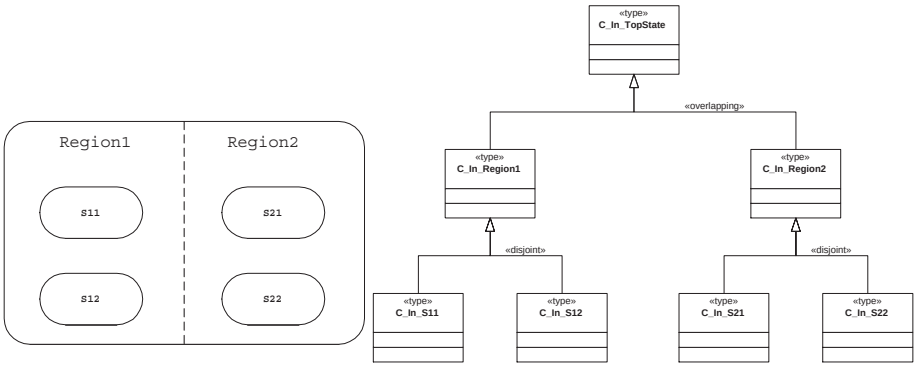
Afin de traduire une spécification UML dans le modèle des (IO)LTS sur lequel sont basés les outils que nous souhaitons utiliser, il est nécessaire de donner une sémantique précise aux constructions d'UML que nous utilisons.

Malheureusement, UML n'est qu'un langage semi-formel. Sa syntaxe abstraite est certes précise (elle est basée sur le méta-modèle de UML, complété avec des contraintes structurelles exprimées en OCL). Cependant, sa sémantique est ambigue, et pour l'essentiel informelle. Beaucoup de travaux de recherche sont actuellement menés afin de donner à UML une véritable sémantique.

La conférence UML'99 [FRA 99] a ainsi donné lieu à plusieurs présentations ayant pour thème principal la formalisation d'UML. Parmi eux, citons [KIM 99] qui propose une formalisation des diagrammes de classes d'UML basée sur le langage Z. Les machines d'états d'UML, fortement inspirées des statecharts de David Harel [HAR 96] sont probablement la partie pour laquelle le plus de travaux ont été réalisés. [PAL 99] donne une sémantique aux statecharts permettant de faire du model-checking à l'aide de Promela/SPIN. [LAT 99] présente une autre approche également basée sur une compilation des statecharts de UML vers Promela. La formalisation des collaborations [ÖVE 99] ainsi que des interactions [KNA 99] est également abordée. On trouve aussi un certain nombre de travaux sur le typage, comme [CLA 99]. Mais il manque toujours à l'heure actuelle une formalisation globale des constructions d'UML, sans doute parce que ces constructions sont difficiles à intégrer dans un cadre formel unifié.

L'approche que nous proposons consiste à transformer la spécification UML initiale en une version équivalente ne faisant usage que de constructions relativement simples de UML (telles les classes et les opérations qui forment la base d'un modèle objet), auxquelles il est plus aisé de donner une sémantique précise. La plupart des transformations concernent les machines d'états, certainement une des constructions de UML les plus complexes. Il est probable que la notion de rôle puisse également s'intégrer dans notre approche, mais cela n'a pas encore été réalisé.

Une machine d'états UML est typiquement constituée d'un ensemble d'états, d'une file d'événements en entrée, ainsi que d'une tâche qui est chargée de prendre les événements de la file un par un et de les traiter selon l'état courant. Nous avons décidé de rendre explicites les différents éléments constituant une machine d'états. Le modèle transformé contiendra ainsi des classes dédiées représentant les files et les tâches de traitement. Les états de la machine peuvent aussi se ramener au concept plus simple de classe : en effet, les transitions sortant d'un sous-état, étiquetées avec un nom d'opération donné, ont priorité par rapport à des transitions sortantes d'un état englobant étiquetées avec la même opération. Cela se traduit tout naturellement en sous-typage. La hiérarchie des états est ainsi traduite en hiérarchie de classes, les transitions surchargées devenant simplement des redéfinitions des opérations correspondantes dans les classes des sous-états. Cette transformation, illustrée par la figure 5, est similaire dans son principe à l'application du *patron de conception* classique appelé *State* [GAM 95].



(a) Machine d'état (b) Classes correspondantes

FIG. 5 –. *Transformation des machines d'états*

L'état courant d'un objet peut évidemment changer après une transition. De plus, du fait de l'existence potentielle de régions concurrentes (les *AND states* dans les statecharts classiques), un objet peut être dans plusieurs états simultanément. Après notre transformation, celà se traduit en classification multiple et dynamique, deux concepts qui existent déjà en UML, même s'ils sont bien souvent ignorés. Notons au passage que cette transformation n'introduit pas de problèmes de "message not understood" typiques de la classification dynamique, car toutes les classes-états possèdent la même interface. Il est donc impossible qu'un objet reçoive un message qui ne soit pas défini dans l'une des classes auxquelles il appartient à un instant donné.

Contrairement aux statecharts classiques où les communications sont synchrones et se font par diffusion globale, les communications entre objets se font en point-à-point et généralement de manière asynchrone en UML, en suivant les liens (canaux de communication) qui relient les objets. Les liens *potentiels* entre objets de chaque classe sont matérialisés par des associations sur les diagrammes de classes, et les liens *effectifs* sont matérialisés sur les diagrammes d'objets ou sur les diagrammes de collaboration. La topologie du réseau d'objets est donc importante, et peut d'ailleurs changer lors de l'exécution du système. Ainsi, la dernière partie des transformations insère dans le diagramme du déploiement initial tous les objets représentant les files de communications rendues explicites dans la topologie. À notre connaissance, seul [PAL 99] prend aussi en compte la topologie au travers des diagrammes de collaboration.

Il existe plusieurs types de communications possibles entre objets, selon qu'un objet est actif ou passif, et selon qu'une requête est bloquante ou non-bloquante. Un objet actif possède son propre flot de contrôle qui traite les requêtes déposées dans sa file séquentiellement. Un objet passif, lui, ne possède pas de flot de contrôle propre, et donc les requêtes qui lui sont faites sont exécutées par le flot de contrôle de l'objet

appelant. UML offre des mécanismes pour synchroniser les requêtes vers des objets passifs afin d'éviter les accès concurrents lorsque cela n'est pas souhaitable (ces mécanismes sont proches sémantiquement du « s nchron i de Java) e Nous ne prenons pas encore en compte ces mécanismes de synchronisation pour objets passifs.

Enfin, le comportement global du système est obtenu en considérant l'exécution en parallèle les flots de contrôle de l'ensemble des objets actifs. Nous avons retenu une sémantique par *entrelacements* pour la construction du graphe d'accessibilité.

3.3. *Le simulateur de spécification UML*

Cette section présente le module de simulation de UMLAUT, dont le but est de traduire une spécification UML en un système de transitions, en s'appuyant sur les transformations présentées précédemment. Partant d'un état global de la spécification, le simulateur propose l'ensemble des transitions tirables depuis cet état.

Les transitions sont étiquetées par les actions *atomiques* exécutées par le système. Une action est toujours exécutée directement ou indirectement par un flot de contrôle précis (appelé "thread"). Si une action appartient à une opération d'un objet passif, elle ne peut être exécutée que si un objet actif lui transmet le contrôle par un appel d'opération ("nested flow of control" en terminologie UML).

Il est important de noter que le contrôle n'est pas forcément initié au sein du système. L'environnement peut également le stimuler, les acteurs étant des objets actifs pouvant envoyer des requêtes à des objets (actifs ou passifs) contenus dans le système. Un système purement réactif n'aura donc aucun comportement spontané en absence de stimulus provenant de l'environnement. Pour pouvoir simuler un tel système de manière exhaustive, il faut donc également simuler un environnement capable d'offrir tous les stimuli possibles au système. On ferme donc le système en faisant des acteurs des *processus chaos*, c'est-à-dire en les munissant d'automates "marguerites" capables de produire tous les stimuli attendus par le système et capables d'accepter toutes les réponses renvoyées par celui-ci. Cela implique notamment de parcourir tout le domaine de valeur des types utilisés en paramètre des messages. Afin d'éviter des divergences inutiles, on supposera l'environnement *raisonnable*, c'est-à-dire qu'un message ne sera envoyé au système que si celui-ci est capable d'y répondre immédiatement. La version actuelle de UMLAUT n'automatise pas la fermeture du système, qui reste pour le moment à la charge de l'utilisateur. Cette contrainte devrait cependant être levée rapidement.

Un état global (encore appelé *configuration*) est donc constitué de :

– l'état de chacun des objets du système
– la topologie du réseau formé par les liens entre objets
– le "locus" de chaque flot de contrôle (et donc la pile d'exécution associée)

Lorsqu'une transition est tirée, un nouvel état global est extrait (par copie profonde de l'ensemble du réseau d'objets). Le simulateur offre aussi une fonction de compa-

raison d'états globaux permettant la détection de cycles dans le graphe d'accessibilité. Cette fonction de comparaison étant définie sur la base des fonctions de comparaison des objets constituants le système (états locaux), il est possible en redéfinissant ces fonctions de comparaison locales de réaliser des abstractions sur le graphes d'accessibilité. L'utilisation d'abstractions pertinentes peut ainsi conduire à une réduction significative de l'espace d'état tout en conservant certaines des propriétés du système.

Le tableau 1 résume les résultats obtenus en utilisant UMLAUT pour construire le graphe d'accessibilité de l'ATC, en bornant la taille des files à 1, puis à 2.

Taille des files	1	2
Nombre d'états	13094	560216
Nombre de transitions	45396	2290713
Taille mémoire nécessaire	74MB	4GB

TAB. 1 –. *Résultat du simulateur sur l'exemple de l'ATC*

La taille importante de mémoire nécessaire pour construire le graphe complet pourrait être réduite d'un facteur nb-transitions/nb-états : Ce facteur est dû au fait que le simulateur n'est pas prévenu lorsque certains états ne sont plus utilisés et ne peut donc pas libérer la mémoire correspondante. Ce problème est sur le point d'être résolu.

3.4. *Limitations actuelles et développements futurs du simulateur*

Le simulateur inclus dans UMLAUT est encore en développement, et souffre donc de quelques limitations.

La première limitation concerne la spécification des actions exécutées par les objets. UML n'offre pas à l'heure actuelle de définition officielle pour les actions. Un groupe s'est formé qui travaille à la définition d'un langage d'actions et à sa formalisation, ce qui permettra à terme de combler cette lacune. Néanmoins, il est possible de pallier ce problème de plusieurs façons :

– en anticipant et en ajoutant ainsi à UML dès maintenant un ensemble limité d'actions très simples que l'on retrouvera forcément dans le futur langage d'actions, comme l'appel d'opérations et l'affectation d'expressions OCL

– en permettant l'utilisation de fragments de code écrits dans un langage de programmation existant (alternative proposée par certains ateliers de génie logiciel).

La seconde possibilité est souvent séduisante, mais les actions ainsi spécifiées restent bien souvent hors de portée des analyses que peut conduire un outil UML ; il est par exemple difficile de s'assurer de l'atomicité de telles actions, ou d'éviter certains effets de bord indésirables. Cette absence d'un langage d'actions bien défini impose certaines hypothèses simplificatrices pour la simulation de modèle UML. Nous considérerons pour la simulation que les transitions des machines d'états de UML sont atomiques, et que les différents objets actifs communiquent de manière asynchrone

par échange de messages. Dans l'absolu, ce n'est pas le cas puisqu'il est possible d'avoir en UML des transitions qui non seulement ne sont pas atomiques mais qui de plus contiennent des actions de synchronisation potentiellement bloquantes, comme les rendez-vous entre objets actifs ou les appels d'opérations gardées (qui mettent en œuvre des verrous implicites). Les hypothèses que nous avons faites permettent de simplifier la simulation en faisant correspondre les transitions dans le LTS avec les transitions des machines d'états des objets actifs du modèle UML de départ.

Enfin, la manière dont les états globaux sont stockés (par copie profonde) n'est pas optimale. En effet, les actions exécutées lorsque l'on tire des transitions ont un impact souvent très localisé (à un objet seulement). De plus, il est courant que la structure du réseau évolue peu (ou même pas du tout), alors que les états locaux des objets ainsi que les files de messages évoluent constamment. Il est donc probable qu'une factorisation des informations communes à plusieurs états apporte des gains importants en mémoire lors de la simulation.

4. Applications

4.1. *Vérification de propriétés*

Dès lors qu'un système atteint une certaine complexité, il n'est guère aisé de déterminer s'il vérifie les propriétés que l'on attend de lui. Ces propriétés sont souvent énoncées de manières informelles lors de la phase de capture des exigences. L'utilisation rigoureuse des assertions, notamment en utilisant OCL, permet d'exprimer un certain nombre d'invariants que le système ne doit pas enfreindre. OCL constitue donc en quelque sorte un premier pas vers l'utilisation de langages formels pour exprimer des contraintes ou des propriétés sur le système. Il est remarquable qu'OCL constitue une des parties les plus formelles d'UML. Cependant, la plupart des exigences portent sur le comportement et les réactions du système et non pas seulement sur son état à un instant donné. L'utilisation de logiques temporelles permet d'exprimer ce type de contraintes, mais elles sont souvent considérées (à tort ou à raison) comme étant particulièrement difficiles à utiliser.

L'apparition d'OCL dans UML offre l'opportunité de changer cet état de fait. OCL est à présent relativement familier aux utilisateurs d'UML, qui sont de plus en plus enclins à l'utiliser en phase de spécification. Certaines méthodes basées sur UML préconisent d'ailleurs d'utiliser intensivement OCL, et participent ainsi à l'adoption d'OCL. C'est notamment le cas de la méthode Catalysis [D'S 98]. Étendre OCL avec des opérateurs de logique temporelle permettrait d'introduire en douceur l'utilisation de la logique temporelle au sein d'une méthodologie à objets basée sur UML, en se basant sur la syntaxe à présent familière d'OCL pour l'expression des propositions atomiques. Quelques travaux comme [RAM 99] vont dans ce sens, mais beaucoup reste encore à faire.

UMLAUT permet cependant dès à présent de détecter les violations de pré et post conditions ainsi que des invariants décrits en OCL classique. En effet, UMLAUT com-

porte un compilateur d'expressions OCL qu'il suffit d'utiliser au sein du simulateur afin que toute violation d'assertion OCL résulte en une transition spéciale étiquetée "violation d'assertion" dans le LTS final.

4.2. *Génération de tests*

Après avoir vérifié certaines propriétés de la spécification, il est important de pouvoir s'assurer qu'une implantation finale du système sera effectivement conforme à sa spécification. L'outil TGV [JÉR 99] peut être utilisé dans ce but. En effet, cet outil permet de générer des cas de tests à partir d'une spécification et d'un *objectif de test*. L'objectif de test guide la génération du cas de test pour produire des cas de tests ciblés, en restreignant l'exploration du graphe d'accessibilité aux seules transitions que l'on souhaite autoriser. La théorie sous-jacente à TGV se fonde sur les systèmes de transitions étiquetées pour lesquels entrées et sorties sont distinguées. Les modèles et algorithmes de TGV sont présentés en détail dans [JÉR 99] et nous ne nous étendrons pas davantage sur cet aspect dans cet article.

Pour pouvoir générer des tests, TGV a donc besoin d'objectifs. La spécification UML du système fournit déjà des objectifs de tests intéressants sous une forme abstraite : les cas d'utilisations. Le cas d'utilisation «prise en charge d'un avion» représentée par la figure 1 peut être décrit plus précisément par une collaboration et un diagramme de séquence représentant les interactions entre l'environnement et le système qui concernent directement la réalisation de ce cas d'utilisation, représentés dans la figure 6. UMLAUT convertit ensuite ces interactions en IOLTS pour TGV (figure 7(a)).

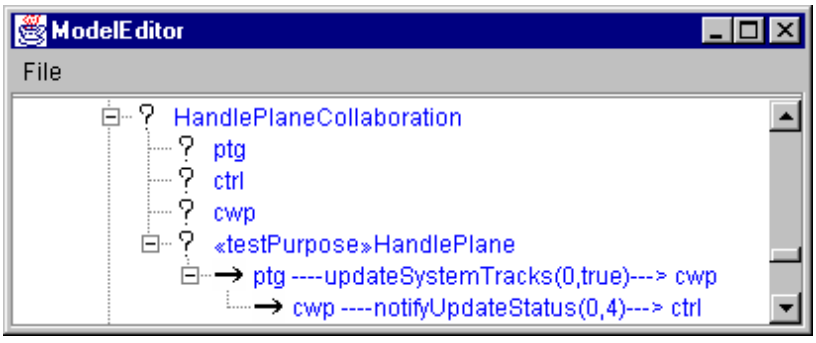
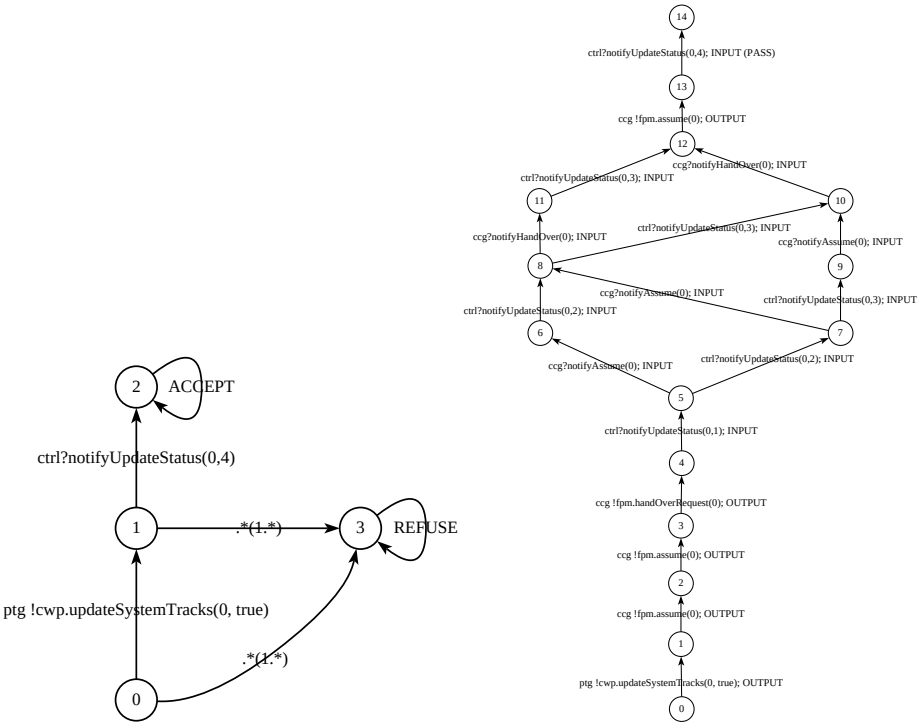


FIG. 6 –. *Objectif de test dans UMLAUT*

Pour obtenir un test, il faut commencer par identifier chaque intervenant (rôle) dans l'interaction parmi les objets du système déployé (voir figure 4). Les points d'interactions entre objets acteurs et objets internes au système deviennent les points de contrôle et d'observation (PCO) du point de vue du testeur. Seuls les messages transitant par ces points sont observables, les messages internes au système n'étant pas accessibles pour le testeur.



(a) Objectif de test (b) Cas de test correspondant

FIG. 7 –. *Génération de test pour l'ATC*

L'interaction dans laquelle on connaît l'identité de tous les objets jouant un rôle constitue alors la trame de base de l'objectif de test tel qu'il est représenté dans la figure 7. Cet objectif de test cherche ainsi à produire un test visant à s'assurer que le système peut bien prendre en charge l'avion numéro 0 (dont le status passe alors à la valeur 4) lorsqu'il rentre dans la zone de l'ATC (événement `date ks ste Trac` signalé au gestionnaire de position `c` par le radar). `tg`

Comme ce n'est qu'une projection partielle du comportement, d'autres événements qui ne concernent pas directement ce cas d'utilisation peuvent éventuellement survenir pendant sa réalisation (les états 0 et 1 de l'objectif de test possèdent tous les deux une transition implicite étiquetée avec "*", c'est-à-dire que tous les événements non-mentionnés sont autorisés).

Les transitions menant dans l'état de l'objectif de test permettent d'élaguer l'exploration du graphe lors de la construction du cas de test, en demandant d'ignorer les événements se référant à d'autres avions (dans le cas d'un déploiement avec deux

avions 0 et 1, l'avion 1 ne nous intéresse pas lorsque l'on teste la prise en charge de l'avion 0). L'exécution de TGV avec cet objectif de test donne automatiquement le cas de test présenté à ses côtés sur la figure 7.

À noter que cette approche du test ne traite pas les données de manière symbolique (les valeurs effectives apparaissent au sein des labels des transitions). D'autres travaux complémentaires comme [OFF 99], basés sur des approches symboliques, abordent le problème de la génération de données pertinentes vis-à-vis d'un critère de couverture des comportements de la spécification.

5. Conclusion et perspectives

Nous avons montré que des méthodes formelles étaient applicables à des spécifications écrites en UML. Nous nous appuyons sur le modèle des systèmes de transitions étiquetées pour donner une sémantique précise et suffisante à un sous-ensemble d'UML pour les techniques que l'on souhaite mettre en œuvre. Le simulateur inclus dans UMLAUT est déjà en mesure d'explorer exhaustivement des graphes de taille respectable et les améliorations évoquées dans la section 3.4 devraient nous permettre d'en repousser les limites. Il peut aussi fonctionner à la volée, lorsque les outils qui lui sont couplés savent exploiter ce mode et peut donc potentiellement travailler sur des graphes infinis (y compris des graphes à branchement infini).

L'utilisation de la boite à outils CADP, et notamment de TGV, permet de produire automatiquement des cas de tests à partir de spécifications UML et d'objectifs de test pertinents. Des travaux sont actuellement en cours pour améliorer l'intégration des méthodes formelles dans UML. Une piste intéressante consiste à poursuivre la formalisation des cas d'utilisation et des collaborations de UML commencée par [ÖVE 99] ainsi que leur lien avec les objectifs de test tels que sait les exploiter TGV, afin d'offrir une aide à la conception des *objectifs* de test, ou mieux encore, de partiellement automatiser cette tâche.

6. Bibliographie

[CCI 87] CCITT, « SDL, Recommendation Z.100 », 1987.

[CLA 99] CLARK T., « Type Checking UML Static Diagrams », FRANCE R., RUMPE B., Eds., *UML'99 - The Unified Modeling Language. Beyond the Standard. Second International Conference, Fort Collins, CO, USA, October 28-30. 1999, Proceedings*, vol. 1723 de *LNCS*, Springer, 1999.

[D'S 98] D'SOUZA D., WILLS A., *Objects, Components and Frameworks With UML: The Catalysis Approach*, Addison-Wesley, 1998.

[FRA 99] FRANCE R., RUMPE B., Eds., *UML'99 - The Unified Modeling Language. Beyond the Standard. Second International Conference, Fort Collins, CO, USA, October 28-30. 1999, Proceedings*, vol. 1723 de *LNCS*, Springer, 1999.

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

[GAR 98] GARAVEL H., « Open/Caesar: An Open Software Architecture for Verification, Simulation and Testing », *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'98)*, vol. 1384, Springer-Verlag, Lecture Notes in Computer Science, 1998.

[HAR 96] HAREL D., NAAMAD A., « The STATEMATE Semantics of Statecharts », *ACM Transactions on Software Engineering and Methodology*, vol. 5, n° 4, 1996, p. 293–333.

[ISO 85] ISO, « LOTOS, A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour », ISO/ DP 8807, March 1985.

[JÉR 99] JÉRON T., MOREL P., « Test generation derived from model-checking », HALBWACHS N., PELED D., Eds., *CAV'99, Trento, Italy*, Springer, LNCS 1633, 1999, p. 108–122.

[JÉZ 99] JÉZÉQUEL J.-M., HO W. M., GUENNEC A. L., PENNANEAC'H F., « UMLAUT: an Extendible UML Transformation Framework », HALL R. J., TYUGU E., Eds., *Proc. of the 14th IEEE International Conference on Automated Software Engineering, ASE'99*, IEEE, 1999.

[KIM 99] KIM S.-K., CARRINGTON D., « Formalizing the UML Class Diagram Using Object-Z », FRANCE R., RUMPE B., Eds., *UML'99 - The Unified Modeling Language. Beyond the Standard. Second International Conference, Fort Collins, CO, USA, October 28-30. 1999, Proceedings*, vol. 1723 de *LNCS*, Springer, 1999.

[KNA 99] KNAPP A., « A Formal Semantics for UML Interactions », FRANCE R., RUMPE B., Eds., *UML'99 - The Unified Modeling Language. Beyond the Standard. Second International Conference, Fort Collins, CO, USA, October 28-30. 1999, Proceedings*, vol. 1723 de *LNCS*, Springer, 1999.

[LAT 99] LATELLA D., MAJZIK I., MASSINK M., « Automatic Verification of a Behavioural Subset of UML Statechart Diagrams Using the SPIN Model-checker », *Formal Aspects of Computing*, vol. 11, 1999, p. 637–664, Springer.

[MEY 92] MEYER B., « Applying "Design by Contract" », *IEEE Computer (Special Issue on Inheritance & Classification)*, vol. 25, n° 10, 1992, p. 40–52.

[OFF 99] OFFUTT J., ABDURAZIK A., « Generating Tests from UML Specifications », FRANCE R., RUMPE B., Eds., *UML'99 - The Unified Modeling Language. Beyond the Standard. Second International Conference, Fort Collins, CO, USA, October 28-30. 1999, Proceedings*, vol. 1723 de *LNCS*, Springer, 1999.

[ÖVE 99] ÖVERGAARD G., « A Formal Approach to Collaborations in the Unified Modeling Language », FRANCE R., RUMPE B., Eds., *UML'99 - The Unified Modeling Language. Beyond the Standard. Second International Conference, Fort Collins, CO, USA, October 28-30. 1999, Proceedings*, vol. 1723 de *LNCS*, Springer, 1999.

[PAL 99] PALTOR I., LILIUS J., « Formalising UML State Machines for Model Checking », FRANCE R., RUMPE B., Eds., *UML'99 - The Unified Modeling Language. Beyond the Standard. Second International Conference, Fort Collins, CO, USA, October 28-30. 1999, Proceedings*, vol. 1723 de *LNCS*, Springer, 1999.

[RAM 99] RAMAKRISHNAN S., MCGREGOR J., « Extending OCL to support Temporal Operators », *Proceedings of the 21st International Conference on Software Engineering (ICSE99) Workshop on Testing Distributed Component-Based Systems (WM3)*, Los Angeles, California, USA, May 1999, ACM press.

[RTF 99] RTF U., *OMG Unified Modeling Language Specification, Version 1.3, UML RTF proposed final revision*, OMG, June 1999.

[WAR 98] WARMER J., KLEPPE A., *The Object Constraint Language: Precise Modeling with UML*, Addison-Wesley, 1998.