REE
REVUE DE L'ELECTRICITE ET DE L'ELECTRONIQUE
N° 3 - Mars 2006
Revue de la Société de l'Électricité, de l'Électronique et des Technologies de l'Information et de la Communication
ENERGIE, TELECOMMUNICATIONS, SIGNAL, COMPOSANTS, AUTOMATIQUE, INFORMATIQUE...

DOSSIER

Le navire tout électrique (1re partie) : concept et applications navire

Une composante essentielle de la stratégie d'action française dans les trente années à venir : les Bâtiments de Projection (BPC) Mistral et Tonnerre.

Processus de fabrication de systèmes répartis centré sur un modèle : l'expérience du projet MORSE

Mots clés

Modélisation,
Vérification,
Géneration automatique de programmes,
Model Checking

■ Frédéric GILLIERS, Fabrice KORDON, Yann THIERRY-MIEG
LIP6-SRC, Université P. & M. Curie

1. Introduction

La complexité des systèmes croît de manière non contrôlée [16] et il apparaît que les techniques « traditionnelles » de développement, basées sur une démarche objet et le test (unitaire puis d'intégration) ne suffisent plus à assurer un besoin de qualité sans cesse croissant. Pour preuve, de la "crise du logiciel" dans les années 1960, on est passé à la "crise *chronique* du logiciel" dans les années 1990 [6].

Le problème est particulièrement crucial dans le contexte actuel d'applications de plus en plus réparties. La répartition génère un asynchronisme entre flots d'exécutions parallèles qui complique la compréhension de la dynamique du système. C'est cette multiplicité des exécutions possibles qui bloque l'analyse des systèmes par des méthodes de tests. Pour s'assurer que le système a un comportement *déterministe* (i.e. il se comporte de manière similaire pour toutes les séquences d'exécutions possible), il faut passer à des approches sûres : les méthodes formelles.

Cependant, les méthodes formelles ne sont hélas pas la solution miracle. De nombreux auteurs comme [23, 17] signalent de nombreuses difficultés comme la difficulté d'apprentissage, l'adaptation à un processus de développement de type industriel ou le passage à l'échelle. Pour rendre ces approches utilisables, il faut être pragmatique et concilier des techniques formelles avec une démarche méthodologique permettant leur usage dans un monde industriel grâce à des outils utilisables par les ingénieurs. Sur ces points, beaucoup de progrès sont fait actuellement.

Dans ce contexte, le projet MORSE définit une démarche méthodologique centrée sur UML pour aider des ingénieurs à construire une application dont le comportement sera déterministe, puis à produire l'application correspondante.

Le projet MORSE couvre donc les points suivants :

1) la mise en place d'une méthodologie adaptée au domaine d'application considéré ;
2) la définition d'un langage de spécification, adapté au domaine et aux besoins de génération de code et de vérification en aval ;
3) la mise en place de techniques de vérification du bon comportement du système ;
4) la réalisation d'un générateur automatique de programmes pour produire le système rapidement et sans dérive par rapport à la spécification.

De plus l'intégration dans un contexte industriel existant impose au niveau modèle une compatibilité avec des

L'ESSENTIEL

La complexité des systèmes croît de manière non contrôlée et il apparaît que les techniques « traditionnelles » de développement, basées sur une démarche objet et le test (unitaire puis d'intégration) ne suffisent plus à assurer un besoin de qualité sans cesse croissant. Dans ce contexte, nous présentons les résultats du Projet MORSE qui vise à intégrer dans une démarche méthodologique centrée sur UML, l'utilisation de méthodes formelles et de générateurs automatiques de programme. MORSE a ainsi pour objectif de rendre accessible à des ingénieurs spécialistes de leur métier des techniques de conception et de développement propres aux systèmes répartis.

SYNOPSIS

As the complexity and size of software systems increases uncontrolably, "traditional" development methodologies that rely on tests to ensure quality needs are showing their limits. In this context, we present the results of the MORSE project, which aims at offering a development methodology based on UML, that offers support for code generation and formal verification. Thus MORSE's goal is to provide software designers expert in their business model with high level concepts adapted to distributed systems, that help to master their complexity.

aspects de l'application modélisés en UML, et au niveau logiciel un mécanisme permettant la réutilisation et l'intégration de composants existants.

Cet article vise à relater l'expérience dans ce domaine du projet MORSE et à en présenter les premiers résultats au bout de deux ans (sur trois) de déroulement du projet.

La section suivante évoque les problèmes du développement d'applications réparties que nous adressons. Nous présentons ensuite la démarche méthodologique de MORSE (section 3) avant d'aborder le langage de spécification que nous utilisons (section 4). Nous exposons alors les techniques de vérification mises en œuvre (section 5) puis enfin les aspects liés à la génération automatique de programmes (section 6).

2. Points clefs

Nous présentons ici les problèmes principalement liés aux ruptures dans le cycle de vie du logiciel et les tendances de la communauté scientifique.

2.1. Rupture du cycle de vie

Le cycle de vie du développement logiciel contient plusieurs ruptures importantes (figure 1).

La première se situe entre la phase d'analyse et la phase de conception. On passe d'un raisonnement en langage naturel à un autre basé sur les principes de l'objet (et s'appuyant en général sur UML). Nous ne nous intéresserons pas dans cet article à cette première rupture mais l'intérêt d'approches à base de logique a été identifié dès les années 1990 [15].

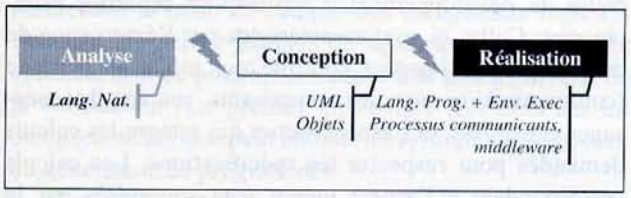


Figure 1. Ruptures dans le cycle de vie.

La seconde rupture, à laquelle nous nous intéressons, se situe entre la phase d'analyse et la phase de réalisation. Là encore, on change de niveau d'abstraction. Pour les systèmes répartis, il s'agit de passer d'une logique de "classification des services" à une logique centrée sur les processus communicants. Cela pose un problème car les choix qui sont envisagés au niveau de la conception peuvent être altérés par une mauvaise implémentation. Cette altération peut engendrer de médiocres performances ou carrément mettre en péril le système.

Dans les systèmes répartis, le passage d'une approche orientée objet à l'implémentation correspondante n'est pas un processus trivial. En effet, les deux logiques sont différentes et pas forcément compatibles. Une première solution consiste à encapsuler la notion d'objet dans une architecture répartie au moyen d'un middleware (on parle alors d'objets répartis). Le rôle du middleware est d'assurer la transposition dans un environnement réparti des mécanismes de base offerts par les langages objets.

Mais cette approche, si elle permet de faciliter le travail de mise en œuvre d'une conception objet, pose d'autres problèmes car les applications ainsi développées restent confinées dans le modèle client-serveur. Ainsi, lorsque le passage à l'échelle de l'application requiert l'usage de techniques dites "pair-à-pair", elle devient rapidement inapplicable.

Une solution plus complète implique la mise en place d'une démarche centrée sur une méthodologie de développement. Une telle démarche doit s'appuyer sur des notations et techniques permettant d'amener les concepteurs à les considérer progressivement. Dans de telles approches, l'outillage (notamment la génération automatique de programmes) joue un rôle crucial. L'usage fréquent de techniques de modélisation peut également permettre (si le modèle est mathématiquement fondé) la mise en place de techniques de vérification dédiées. Ce point est très important pour des systèmes complexes (comme les systèmes répartis) car les approches classiques basées sur le test atteignent leurs limites [17].

Industriels et universitaires essayent de faire progresser l'état de l'art au moyen de projets institutionnels. Pour "simplifier" la tâche, on s'intéresse actuellement à des domaines d'applications particuliers, pour lesquels les besoins sont cruciaux. Par exemple, le projet RNTL COTRE [1] propose une démarche centrée sur le temps réel dans les systèmes avioniques. Il s'appuie sur les standards du marché (UML, HOOD) ainsi que sur l'outil Scade qui a déjà fait ses preuves dans le domaine de cible. L'objectif est de garantir des propriétés temps réel dans le système.

Dans un domaine similaire mais centré sur les problèmes de répartition, le projet RNTL MORSE [25] s'intéresse à l'expérimentation des méthodes formelles dans un contexte industriel différent : celui des applications à composants répartis dans un environnement asynchrone (COTRE se concentre sur les approches synchrones).

L'objectif du projet MORSE est de fournir une solution pour le développement d'applications industrielles certifiables, notamment pour des systèmes de drones (avions sans pilotes). Nous nous intéressons en particulier aux problèmes liés aux relations entre le vecteur (l'avion sans pilote) et le système de contrôle au sol. Les aspects temps réels sont moins critiques que pour l'applicatif de pilotage automatique situé sur le vecteur. Par contre, les exigences sur la fiabilité du comportement (les protocoles d'interaction entre le vecteur et le sol) sont importantes. La situation est rendue délicate par l'exécution dans un

environnement réparti ne disposant que de communications asynchrones (liaisons hertziennes).

En outre, sans traiter directement ce point dans le cadre de MORSE, il faut pouvoir ensuite certifier le système produit en suivant les techniques en vigueur dans le domaine considéré. MORSE n'envisage pas d'intégrer le processus de certification dans la méthodologie mais plutôt de l'appliquer aux résultats produits.

2.2. Développement orienté modèle

Notre méthodologie est basée sur un cycle de développement centré sur le *modèle* de l'application (on parle de *Model-based development*). L'intérêt principal de ce type d'approche est de gagner en abstraction et de s'affranchir partiellement des contraintes liées à l'environnement utilisé pour le déploiement (langages d'implémentation, technologies particulières, middlewares ...).

UML propose une notation uniformisée pour la modélisation d'applications. Sa vocation est d'être suffisamment large pour couvrir le spectre complet des applications logicielles. Le choix de ce langage de conception rend l'interprétation des diagrammes de la norme ambiguë, certains aspects (au mieux décrits comme des « points de variation sémantique ») étant laissés à la discrétion de l'utihsateur. Ces imprécisions rendent difficile l'exploitation automatique d'UML dans le contexte des systèmes répartis, en particulier pour l'analyse du comportement ou la génération de programmes respectant la sémantique (mal définie !) du modèle. De fait, les ateliers UML présents sur le marché proposent de la génération automatique de code limitée aux squelettes des classes définies dans le modèle, la partie comportementale n'étant pas ou peu abordée.

Les travaux entrepris dans le cadre de MDA (*Model Driven Architecture*) [19] ont pour objectif de lever une partie de ces limitations. Cette démarche propose plusieurs étapes pour affiner le modèle jusqu'à rendre la génération de code possible. Néanmoins l'implémentation de cette démarche n'est pas encore complète ; elle se heurte également aux limitations de la sémantique d'UML dans le contexte des applications réparties.

Des méthodologies telles que Fondue [21] tentent de lever ces limitations. Fondue est basée sur l'utilisation de profils pour lever les limitations d'UML dans le domaine de la répartition. Néanmoins, cette approche ne s'intéresse pas à la vérification du modèle vis-à-vis de sa spécification. Cela vient du fait que le comportement des composants ne sont pas pris en compte. De même, seul le code d'initialisation du middleware et les interfaces des composants sont générés. Le comportement attendu des composants doit donc être implémenté de manière "traditionnelle" ce qui limite la productivité et n'adresse pas le problème de la dérive entre la spécification et le code produit.

3. Méthodologie de développement

L'objectif de nos travaux est de supprimer les ruptures apparaissant entre les étapes du cycle de développement. Pour cela, nous proposons une approche orientée modèle visant la production automatique du programme correspondant à l'application.

3.1. L'approche proposée par MORSE

L'approche proposée dans le cadre du projet MORSE est basée sur la vérification formelle (pour vérifier que le modèle de l'application respecte les exigences définies par la spécification) et la génération de programmes (qui assure l'implémentation du modèle sans dérive par rapport au comportement spécifié).

Compte tenu de l'environnement industriel du projet, la spécification de l'application doit être réalisée en UML qui est le standard de fait de l'industrie pour la modélisation.

Néanmoins, les limitations de la sémantique UML ne nous permettent pas d'utiliser des techniques de génération de programmes et de vérification formelle directement sur cette spécification. Nous introduisons donc le langage L*f*P qui a pour objectif de formaliser une partie de la représentation de l'application répartie. La méthodologie que nous proposons dans le cadre du projet MORSE est illustrée sur la figure 2 que nous détaillons dans le reste de cette section.

Une méthodologie dédiée au domaine d'application

Pour être efficace, nous devons tenir compte du domaine d'application visé. Nous nous plaçons dans le cadre du développement d'applications réparties asynchrones. Celles-ci sont caractérisées par l'émergence de deux aspects : l'aspect contrôle qui pilote le système (communication entre les composants, routage des messages échangés), et l'aspect métier qui intègre les calculs demandés pour respecter les spécifications. Les calculs correspondant à l'aspect métier sont supervisés par le code correspondant à la partie contrôle.

En appliquant une méthode d'analyse adaptée, les aspects métiers peuvent être ramenés à des filtres invoqués par la partie contrôle de l'application. Il existe peu d'interactions entre ces traitements ; elles peuvent alors s'exprimer sous la forme de données transitant d'un traitement à l'autre.

Ainsi, la première étape de la méthodologie est le développement d'une spécification de haut niveau écrite en UML. Cette spécification doit mettre l'accent sur la séparation entre les aspects *contrôle* et *métier*.

L'intégration entre le langage L*f*P et la notation UML est réalisée par la définition d'un profil UML spécifique pour le développement d'applications réparties. Sa définition a été réalisée en collaboration avec la société Aonix, partenaire du projet MORSE [25]. L'objectif de ce profil est d'obtenir une modélisation précise de la struc-

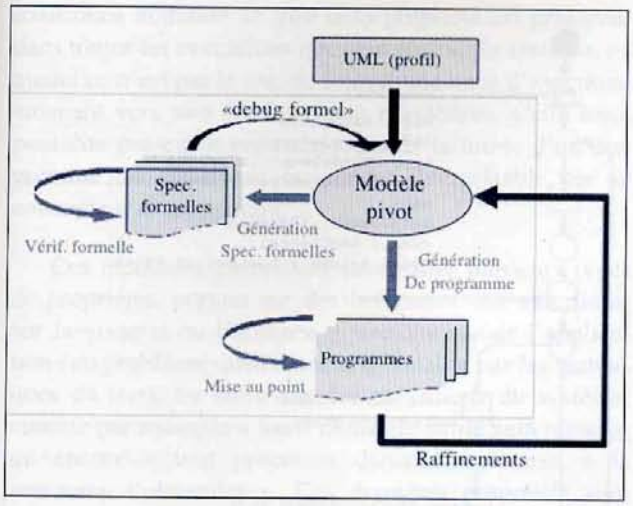


Figure 2. Méthodologie associée à L/P.

ture de l'application, et de fournir une spécialisation de la sémantique d'UML permettant sa traduction automatisée en L/P par des techniques de transformation de modèle. Cette traduction s'appuie sur le méta-modèle du langage L/P [18] réalisé avec MOF [12].

L/P, le langage de spécification pivot

Le langage L/P [22, 10] permet de spécifier le comportement du système à l'aide de processus séquentiels communiquant via des *médias*. Ce langage place des restrictions sur le comportement associé à des classes UML (comme l'absence de parallélisme intrinsèque) qui permettent de préciser la sémantique d'exécution du modèle.

Il est essentiel à ce niveau de fournir des mécanismes permettant la prise en compte des composants logiciels existants qui seront intégrés au squelette de contrôle du système, afin de pouvoir les traiter à la fois au niveau de la vérification (i.e. prendre en compte leur effet sur les comportements que peut adopter le système) et au niveau du générateur de programmes.

Ces composants logiciels constituent la partie métier de l'application et sont appelés composants externes car ils ne sont pas directement adressés par notre méthodologie. Ils n'interviennent pas directement dans le contrôle du système, ce qui nous permet de focaliser l'attention des développeurs sur la partie contrôle de l'application qui comporte les difficultés spécifiques du développement d'applications réparties. Les interactions entre les composants de contrôle et les composants externes seront réalisées par des *appels externes*, c'est-à-dire des appels aux méthodes des composants externes depuis les composants de contrôle.

Ces composants externes peuvent être :

- des composants provenant d'une version antérieure de l'application ;
- des composants sur étagère fournis par un sous-traitant ;
- de nouveaux composants développés pour l'application, mais au moyen d'une autre méthode de développement.

L/P sert de modèle pivot pour la vérification formelle des propriétés attendues de l'application, et la génération de programmes.
Il définit de manière non-ambiguë :

- le comportement des composants assurant le contrôle du système ;
- l'interface avec les composants métiers ;
- l'interaction entre le contrôle du système et les composants métier.

Le langage pivot est fondamental pour la méthodologie car il isole la partie contrôle de l'application et la représente au moyen d'une notation adaptée aux applications réparties. Le langage L/P définit donc une sémantique sous-jacente pour les diagrammes dynamiques UML définissant les interactions entre les composants ; cette approche est très similaire à [3] en utilisant une autre base formelle.

Vérification formelle

La vérification formelle permet d'analyser le modèle L/P de l'application d'identifier les comportements déviants par rapport aux propriétés attendues. Ce travail permet par exemple d'assurer que les protocoles de communication développés respectent le comportement prévu. L'objectif est de pouvoir déboguer formellement l'application au niveau du modèle pivot.

Nos techniques de vérification sont basées sur le model checking, qui grâce à son caractère automatique demande une moins grande expertise des utilisateurs et supporte mieux un cycle de vie itératif dans lequel le modèle est soumis à des évolutions fréquentes. Les techniques utilisées sont détaillées en section 5.

Génération automatique de programmes

Lorsque le modèle L/P correspond aux exigences exprimées, un générateur produit automatiquement une implémentation de la spécification dans l'environnement cible de l'application. Les programmes générés correspondent à la partie contrôle de l'application et s'interfacent avec la partie métier. Les aspects techniques de la génération de programmes sont abordés à la section 6.

4. Le langage L/P

Ce langage est le point central de notre méthodologie de développement. Il est utilisé pour représenter la partie contrôle de l'application répartie à développer. La sémantique complète du langage L/P est présentée dans [7] ; celui-ci propose deux vues complémentaires du système :

- la vue *architecturale* définit les composants de l'application, ainsi qu'une représentation statique de leurs communications ;

- la vue *comportementale* définit le comportement des composants de contrôle de l'application.

4.1. La représentation architecturale

La représentation architecturale est réalisée par le diagramme d'architecture du modèle.
Celui-ci définit trois types de déclarations :

- les composants de l'application ;
- les liens de communication entre ces composants ;
- les déclarations globales du modèle.

On distingue deux types de composants dans L*f*P : les classes et les *médias*. Les *binders* assurent la transmission des messages entre les composants du modèle. Les *classes* sont les composants applicatifs du modèle, elles assurent les liaisons entre le système et les composants externes. Les *médias* réalisent les protocoles de communications utilisés entre les classes en assurant le routage des messages échangés.

La figure 3 est un exemple de diagramme d'architecture pour un système simple. La classe *DisplayManager* utilise un ensemble de serveurs de calculs implémentés par la classe *Server* pour calculer une courbe de Mandelbrot. Le routage des messages entre les classes est assuré par le média `MsgChannel`. L'interface entre les classes et les médias est assurée par les binders qui représentent les files de messages. Le type `GUI` représente un composant externe assurant l'affichage du résultat. C'est un type abstrait qui ne sera manipulé que via les méthodes qui lui sont associées. Il fait partie de l'aspect métier de l'application, son comportement n'est donc pas explicité par le modèle L*f*P.

4.2. La représentation comportementale

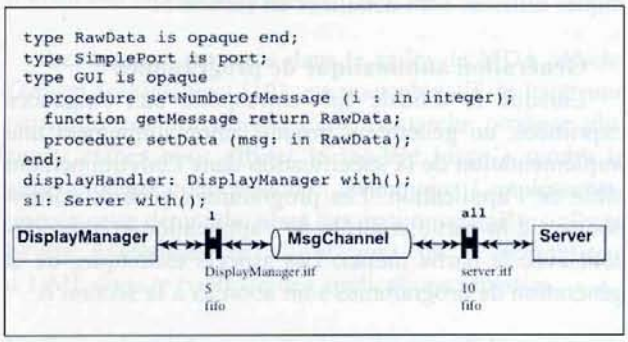


Figure 3. Exemple de diagramme d'architecture.

La vue comportementale est réalisée par les diagrammes de comportement des composants du modèle. Ils définissent le comportement dynamique de chaque composant pendant son exécution. La structure de ces diagrammes correspond à un automate état / transition. Parmi les instructions disponibles, on distingue :

- les instructions de structuration (boucles, tests...) ;
- les instructions de communication (envoi et réception de messages) ;
- l'instanciation dynamique de composant.

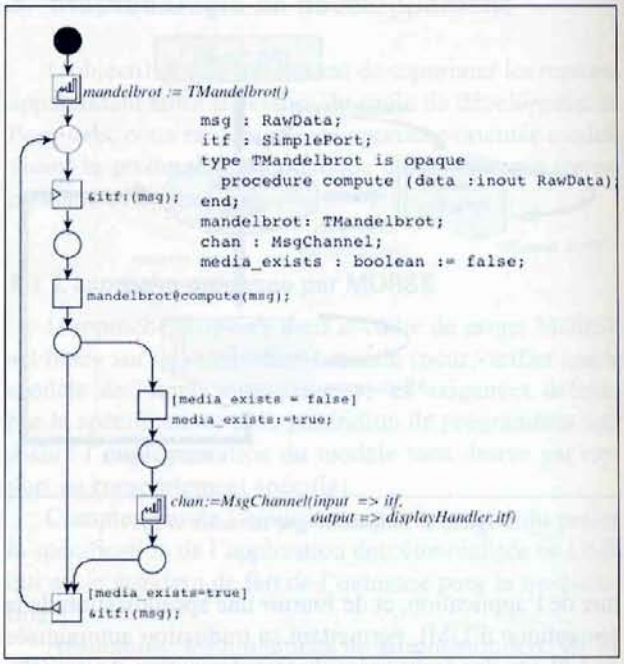


Figure 4. Comportement de la Classe `server`.

La figure 4 présente le diagramme de comportement de la classe `server`. Cette classe commence par construire une instance du composant externe `TMandelbroot` chargé de réaliser les traitements *métiers*. Ensuite, elle attend un message sur le *binder itf* et appelle la méthode *compute* sur le composant externe. Enfin, la valeur de retour doit être transmise au client, pour cela, on instancie le média nécessaire (la première fois) ou on l'utilise directement. Le composant se remet alors en attente du prochain message.

5. Vérification formelle

Il existe deux classes de techniques de vérification formelles : la *preuve* et le *model checking*. Pour la première, le système à vérifier est vu comme un ensemble d'axiomes qui servent à prouver une propriété. Cette technique n'est pas automatique, des outils d'aide à la preuve existent mais demandent une forte expertise. En revanche, la démonstration peut être paramétrée.

Notre choix s'est porté sur le model checking pour les raisons que nous présentons ici.

5.1. Le model checking

Nous proposons de vérifier le comportement du modèle L*f*P par model checking, ou exploration exhaustive de l'espace d'états à la recherche de comportements s'écartant de la spécification. Le gros avantage de cette technologie sur les systèmes de preuve est que les outils de model checldng s'utilisent en "push-the-button". Etant donné un modèle et une propriété, l'outil est capable *sans*

assistance humaine de dire si la propriété est préservée dans toutes les exécutions que peut réaliser le système, ou quand ce n'est pas le cas, de fournir une trace d'exécution amenant vers une configuration nondésirée. Cette trace peut être présentée par exemple sous la forme d'un diagramme de séquence, facilement interprétable par le concepteur d'application.

Ces méthodes permettent de vérifier plusieurs types de propriétés, portant sur des invariants, des assertions, sur la vivacité ou l'absence d'interblocage de l'application (un problème difficilement délectable par les techniques de test), ou enfin sur des exécutions du système, comme par exemple « toute demande émise sera reçue », ou encore « tout processus demandant l'accès à la ressource l'obtiendra ». Ces dernières propriétés sont généralement spécifiées à l'aide d'expressions de logique temporelle.

Limites du model checking

Mais ces techniques ne sont pas une panacée et leur mise en œuvre dans une chaîne de développement industriel est coûteuse et difficile. Le premier obstacle vient de la complexité des formalismes utilisés comme support de la vérification. Le coût de formation des ingénieurs à leur usage est trop élevé, pour un retour difficile à évaluer à cause de la rupture qui existe entre le modèle vérifié et l'application effectivement déployée. Ce point est au moins partiellement pris en compte par la méthodologie proposée, qui suppose l'utilisation transparente de la vérification autour de la notation modèle L*f*P.

De plus, l'exploration des états accessibles d'un système se heurte à un problème d'explosion combinatoire, en particulier dû à l'entrelacement des exécutions dans un contexte réparti. Pour lutter contre ce problème, il existe deux familles de techniques : la première est basée sur l'agrégation d'états en classes d'équivalences, et la deuxième sur une compression de la représentation de l'espace d'états.

5.2. Lutte contre l'explosion combinatoire de l'espace d'états

Dans le cadre du projet MORSE, nous avons testé la mise en œuvre de techniques issues de ces deux familles, qui sont présentées ici. Les deux techniques que nous avons sélectionnées cumulent leurs effets et permettent de gagner plusieurs ordres de grandeur dans la complexité des systèmes que l'on peut traiter.

Exploitation des symétries du système

L'agrégation d'états en classes d'équivalence est une technique qui consiste à regrouper des états indistinguables *du point de vue d'une propriété* à vérifier. Par exemple, une variable qui représente un capteur dont le domaine est continu, comme un altimètre, doit être discrétisée pour permettre sa représentation ; cela produit en général un très grand domaine pour la variable qui fait exploser la vérification. Mais bien souvent seules certaines valeurs seuils sont significatives du point de vue contrôle (i.e. il est interdit d'ouvrir le train d'atterrissage à plus de 40 mètres d'altitude). L'abstraction consiste ici à grouper tous les états tels que l'altitude est inférieure (ou respectivement supérieure) à la borne identifiée.

Un deuxième exemple de symétries du système exploitables en model checking est le suivant : les systèmes distribués sont souvent composés d'instances équivalentes du point de vue comportement mais qui diffèrent par leur identité, par exemple des processus exécutant le même code mais ayant chacun un *pid* différent, ou une classe représentant des adresses mémoires, si l'on considère que toutes les adresses sont finalement équivalentes.

Dans les cas favorables, où le système présente des symétries de ce type, il est possible de construire un graphe des états accessibles *exponentiellement* plus petit que le graphe complet, qui permet cependant de vérifier la propriété. Pour permettre la mise en œuvre automatisée de ces techniques, nous avons développé un outil d'analyse de la spécification [26], qui permet de détecter entièrement automatiquement les bornes significatives pour une propriété et de les exploiter par la construction d'un graphe quotient dans lequel les nœuds sont des classes d'équivalence. Cette technique a déjà été expérimentée avec succès dans le cadre du projet PolyORB [11].

Utilisation d'une représentation compacte

La deuxième classe de techniques que nous avons mise en œuvre pour la vérification dans MORSE est basée sur une représentation extrêmement compacte de l'espace d'états du système. Nous utilisons pour cela une variante des diagrammes de décision binaires (BDD) [2] appelée Data Decision Diagram (DDD) introduits en 2002 [4]. Cette structure d'arbre partagé exploite le fait que les états accessibles d'un système ont une variance limitée, par exemple la seule différence entre les états E1 et E2 va être la valeur d'une variable parmi 500. Ce type de structures s'appuient sur un cache d'opérations et une table d'unicité pour offrir des manipulations dites « symboliques », où l'on manipule des ensembles d'états et non des états individuels.

Les BDD classiques ont montré leur puissance dans le domaine de la vérification matérielle, qui sont en effet particulièrement adaptés à la représentation de fonctions booléennes comme les circuits logiques. Cependant, l'application de ces méthodes aux systèmes logiciels se montre plus difficile en général. Grâce à la grande flexibilité offerte par les DDD, qui supportent des types entiers, et un mécanisme particulier pour la définition des opérations appelé *homomorphismes*, il a été possible de définir directement les opérations de base de L*f*P de façon symbolique. Nos expérimentations [14] ont montré que cette technique permet effectivement la

vérification de modèles de taille conséquente.

Travaux en cours

Nous avons de plus récemment développé des extensions *hiérarchiques* au modèle des DDD [5], qui sont particulièrement bien adaptés à la vérification de modèles obtenus par composition de sous-modèles. Les performances exceptionnelles de ce nouveau modèle de représentation nous permettent une bonne exploitation de la définition modulaire et hiérarchique d'une spécification de haut niveau.

Enfin nous avons montré qu'il est possible de cumuler ces deux familles de techniques, en construisant un graphe de classes d'équivalence directement sur une représentation symbolique en DDD [27]. Ce travail n'est actuellement qu'à l'état de prototype, mais les résultats déjà obtenus sont très prometteurs.

L'utilisation de ces techniques de pointe est nécessaire pour espérer traiter le problème de vérification en passant à l'échelle, le nombre d'états accessibles d'un système dépassant fréquemment 10^{20}, ce qui rend impossible une approche naïve par énumération. Les techniques actuellement développées permettent de supporter des espaces d'une taille de 10^{40} à 10^{500} états sur des problèmes favorables.

6. Génération automatique de programmes

Si la génération de processus séquentiels est un domaine bien connu, la réalisation du générateur de programme pour des applications réparties pose quelques problèmes délicats :

- il faut intégrer les composants logiciels au squelette de contrôle généré par le système ;
- il faut définir les caractéristiques d'un exécutif capable de supporter la sémantique comportementale de L*f*P ;
- il faut identifier les mécanismes qui permettent de déployer les différents composants sur l'architecture cible (les machines au sol, le code embarqué sur le vecteur).

Nous verrons en section 6.4 que les deux derniers points imposent des contraintes sur le middleware support de l'exécutif associé au langage L*f*P.

6.1. Processus de génération de programmes

Le processus de génération de programmes est réalisé en deux étapes illustrées sur la figure 5 :

- l'adaptation du modèle à la sémantique d'exécution de la plate-forme cible ;
- l'écriture des instructions correspondantes dam le langage cible.

La première étape est appelée "sémantique". Elle assure la traduction des instructions définies dans le langage de modélisation vers les instructions disponibles sur la plate-forme cible. L'objectif est de fournir une représentation

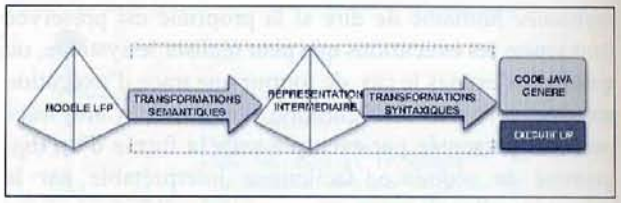


Figure 5. Etapes de génération de programmes.

structurée d'un PSM de l'application. La représentation obtenue dépend de l'environnement de déploiement (par exemple CORBA), ni d'une implémentation spécifique la plate-forme, ni du langage cible.

La deuxième étape est appelée "syntaxique". Elle permet d'intégrer les spécificités du langage et de l'environnement de déploiement. Elle produit un ensemble des fichiers source implémentant le modèle. Le programme obtenu implémente la partie contrôle de l'application. Le lien avec les composants externes est assuré grâce à la définition de leur interface sous forme de types opaques L*f*P.

Par exemple, si la plate-forme cible est CORBA, la première étape fournira un modèle basé sur la sémantique des appels de méthodes distantes définies dans la spécification CORBA. La deuxième étape réalisera les ajustements nécessaires à l'implémentation de CORBA effectivement utilisée lors du déploiement.

6.2. Architecture des prototypes générés

Le générateur de programmes s'appuie sur une architecture type présentée en figure 6. Cette architecture intègre les aspects contrôle (définis en L*f*P) et les aspects métiers (exprimés à l'aide de types opaques et pour lesquels une implémentation est disponible).

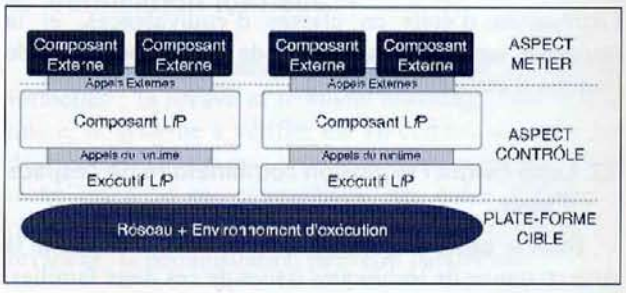


Figure 6. Structure d'une application générée.

La partie centrale de l'application gère le contrôle du système. Les composants qui l'implémentent sont entièrement générés à partir de la spécification L*f*P et implémentent la sémantique du langage. Ainsi, à chaque instruction L*f*P correspond une règle de génération lui associant un gabarit de code dans cette architecture. L'ensemble des règles de génération nécessaires pour implémenter le langage L*f*P est défini dans [8].

L'interface avec les composants métiers est impléd-

mentée par des appels aux méthodes de ces composants. La conformité est assurée par la définition des types opaques correspondants. Notons qu'à partir de leur définition, une interface vide peut être générée pour faciliter l'adaptation des composants préexistants au système en cours de développement.

L'exécution des composants L*f*P est supportée par un exécutif qui assure :

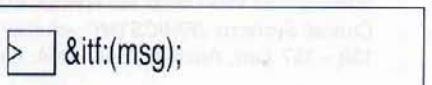
- la liaison entre les composants de la spécification et la plate-forme d'exécution ;
- l'implémentation des types de base du langage ;
- la gestion des ressources systèmes requises par l'exécution des composants L*f*P.

L'interface de cet exécutif fournit un environnement utilisé par le générateur de programmes pour accéder aux fonctions de l'environnement d'exécution (le système d'exploitation et/ou le middleware utilisé) [10].

6.3. Extraits de programme généré

Cette section présente deux exemples très succincts de code généré. Ils donnent une idée générale de l'interface de l'exécutif requis pour implémenter la sémantique du langage L*f*P.

La figure 7 présente une transition issue de la figure 4 ainsi que le code généré correspondant. La transition attend un message sur le *binder* `itf`, le contenu du message est stocké dans la variable `msg`. Cette instruction est réalisée par la fonction `getSimpleMessage` dans l'exécutif L*f*P. Le contenu du message (ici une seule valeur) est ensuite copié dans la variable `msg`.




```
msg = runtime.getSimpleMessage(ITF);
MSG = (RAWDATA) msg.getParameter( 1);
```

Figure 7. Réception de message en L*f*P *et le code généré associé.*

La figure 8 présente une transition d'instanciation dynamique d'un média L*f*P (le même mécanisme permet d'instancier des classes). Elle est également issue de la figure 4. On instancie ici le média `MsgChannel` nécessaire à l'envoi du message de retour vers le client. Les variables `attr_list et initial_values` contiennent respectivement la liste des attributs à initialiser, et les valeurs à leur associer. L'appel à `newInstance` construit l'instance du nouveau composant, lui alloue les ressources nécessaires pour son exécution et l'insère dans les structures de données de l'exécutif. La valeur retournée est la référence du nouveau composant dans le service de nommage, elle peut donc être transmise à des composants distants.

Les exemples présentés ci-dessus ont été produits par


```
String[] Attr_List19 = {"INPUT", "OUTPUT"};
Object[] inital_values20 = {ITF,
 GlobalDeclarations.DISPLAYHANDLER.getBinder("ITF")};
CHAN = runtime.newInstance("MSGCHANNEL", Attr_List19,
                           inital_values20);
```

Figure 8. Instanciation dynamique d'une classe L*f*P *et le code généré associé.*

un générateur de programmes prototypes en langage Java [8]. Pour des besoins d'expérimentation du projet MORSE, un générateur vers C++, reprenant les principaux choix du générateur Java, est en cours de finalisation.

6.4. L'environnement d'exécution

Une fois l'application répartie développée, il reste à la déployer sur un ensemble de machines. Dans le cadre d'un projet comme MORSE, cela signifie que certains processeurs peuvent être embarqués. Cette opération pose donc des contraintes particulières.

L'usage d'un middleware permet de s'abstraire des caractéristiques du système d'exploitation et du matériel. Mais cette approche a l'inconvénient d'entraîner un important surcoût, que ce soit en emprunte mémoire ou en utilisation des ressources.

Ainsi, le middleware "idéal", dans le contexte de MORSE, doit tout d'abord être fortement configurable : il doit fournir des services pour implémenter l'exécutif L*f*P aussi bien sur une machine équipée d'un système d'exploitation complet (par exemple sous Linux) que sur des nœuds ayant de fortes contraintes mémoire.

Une grande versatilité sur les API proposées aux développeurs et les protocoles d'interaction entre nœuds d'une application répartie serait aussi un avantage complémentaire certain. En effet, cela permet d'adapter les protocoles en fonctions des conditions dans un système réparti hétérogène. Par exemple, dans un système partiellement embarqué dans des satellites, on choisira un protocole économe en bande passante pour une liaison espace/sol et des protocoles plus lourds entre les machines du centre de calcul qui exploite les données satellitaires.

De tels middleware commencent à exister. Notons dans ce domaine les projets TAO [28], Quarterware [24] ou PolyORB [20]. TAO est utilisé dans de nombreux projets, et PolyORB, outre une utilisation dans plusieurs projets, fait l'objet d'un support technique par une société commerciale. Cela montre qu'une génération de middleware nouvelle intéressante pour supporter notre méthodologie de développement est opérationnelle.

7. Conclusion

Nous avons présenté dans cet article une méthodologie pour les systèmes logiciels répartis développés dans le cadre du projet MORSE. Ce projet regroupe deux partenaires industriels (Sagem-Défense et Aonix) avec deux laboratoires de recherche (LIP6 et LaBRI). Ce projet RNTL a débuté en juillet 2003 et se, terminera en 2006.

La principale caractéristique de la méthodologie, MORSE est d'établir une liaison entre un niveau de description UML (contraint par un profil dédié) et la vérification formelle d'une part et la génération automatique de programmes d'autre part. Nous espérons ainsi rendre les techniques formelles utilisables dans le contexte d'un développement industriel. Des outils intègrent la majeure partie du savoir faire dans la maîtrise des applications réparties, les ingénieurs de développement se focalisent alors sur les parties métier de leur domaine.

La méthodologie est actuellement opérationnelle :

- le profil UML-MORSE est intégré dans une version étendue d'Améos, l'AGL d'Aonix ;
- le langage L*f*P sert de pivot entre la modélisation, la vérification et la génération automatique de programmes. L*f*P est synthétisé automatiquement par Améos ;
- un outil de model checking est en cours d'implémentation, après que des expérimentations sur des spécifications caractéristiques aient démontré la faisabilité du problème [9] ;
- des générateurs de code ont été développés en Java et en C++. Au contraire de la plupart des produits basés sur UML, ils produisent l'application complète (structure et comportement) dans un contexte d'exécution repartie.

Les modèles et les outils ont été éprouvés sur des cas d'études simples. Dans la dernière année du projet MORSE, nous allons réaliser une étude de cas complexe issue des activités de Sagem-Défense. Cette étape de validation nous permettra d'affiner les besoins et d'identifier les étapes futures à mettre en œuvre pour la conception d'un environnement de développement industriel basé sur les principes du *prototypage par raffinement* [13], un cycle de développement itératif centré sur un modèle de haut niveau.

Références

[1] B. Berthomieux, P.-O. Ribet, F. V. .J.-L. Bernartt, J.-M. Farines, J.-P. Bodeveix, M. Filali, G. Padiou, P. Michel, P. Farail, P. Gauffilet, P. Dissaux, & K.-L. Lambert, "Towards the Verification of Real-Time Systems in Avionics: the Cotre Approach". In *Proceedings of the 8th International Workshop on Formal Methods for Industrial Critical Systems (FMICS'03)*, 2003.

[2] R. Bryant, "Graph-based Algorithms for Boolean Function Manipulation", *IEEE Transactions on Computers*, 35(8) : 677–691, August 1986.

[3] R. Clark & A. Moreira, "Use of E-LOTOS in Adding Formality to UML", *Journal of Universal Computer Science*, 6 (11) : 1071–1087, 2000.

[4] J.-M. Couvreur, E. Encrenaz, E. Paviot-Adet, D. Poitrenaud & P.-A. Wacrenier, "Data Decision Diagrams for Petri Net Analysis". In *Proc. of ICATPN'2002*, volume 2360 of *Lecture Notes in Computer Science*, pages 101–120. Springer Verlag, June 2002.

[5] J.-M. Couvreur & Y. Th.-Mieg, "Hierarchical Decision Diagrams to Exploit Model Structure". In *Proc. of the 25th IFIP WG 6.1 Int. Conf. on Formal Techniques for Networked and Distributed Systems (FORTE'05)*, volume 3731 of *LNCS*, pages 443–457, Taipei, Taiwan, October 2005. Springer.

[6] W. Gibbs, "Software's Chronic Crisis". *Scientific American*, 271(3) :72–81, Sept. 1994.

[7] F. Gilliers, "*Sémantique du langage* L*f*P". Laboratoire d'Informatique de PARIS VI, 2004.

[8] F. Gilliers, "*Développement par prototypage et génération de code à partir de LfP, un langage de modélisation de haut niveau*". PhD thesis, Universite Pierre et Marie Curie (Paris VI), 2005.

[9] F. Gilliers, F. Bréant, D. Poitrenaud & F. Kordon, "Model Checking of High-Level Object Oriented Specifications : The Lf P Experience". In *Third Workshop on Modelling of Objects, Components, and Agents (MOCA'04)*, 2004.

[10] F. Gilliers, J.-P. Velu & F. Kordon, "Generation of Distributed Programs in their Target Execution Environment". In *Fifteenth IEEE International Workshop on Rapid System prototyping*, 2004.

[11] J. Hugues, Y. Thierry-Mieg, F. Kordon, L. Pautet, S. Baarir & T. Vergnaud, "On the Formal Verification of Middleware Behavioral Properties". In *Proceedings of the 9th International Workshop on Formal Methods for Industrial Critical Systems (FMICS'04)*, volume ENTCS 133, pages 139 – 157, Linz, Austria, Sept. 2004. Elsevier.

[12] ISO. Meta Object Facility (MOF) specification. Technical report, ISO, april 2002.

[13] F. Kordon & J. Henkel, "An Overview of Rapid System Prototyping Today". *Design Automation for Embedded Systems*, 8(4) :275–282, december 2003.

[14] F. Kordon & M. Lemoine, editors, "*Formal Methods for Embedded Distributed Systems How to Master the Complexity*". Kluwer Academic, 2004.

[15] L. Lamport. Managing proofs. In T.Margaria & B. Steffen, Editors, TACAS, volume 1055 of *Lecture Notes in Computer Science*, page 34. Springer, 1996.

[16] N. Leveson, "Software Engineering : Stretching the Limits of Complexity". *Communications of the ACM*, 40(2): 129–131, 1997.

[17] Luqi & J. Goguen, "Formal Methods: Promises and Problems". *IEEE Software*, 14(1) :73–85, January / February 1997.

[18] S. N. Mejia, "*Le méta-Modèle LfP*". Laboratoire d'Informatique de PARIS VI, 2004.

[19] OMG, "*Model Driven Architecture (MDA)*". *Document number ormsc/2001-07-01*, 2001.

[20] L. Pautet & F. Kordon, "Des vertus de la schizophrénie pour le prototypage d'applications à composants intéropérables".

TSI, 23(10) :1301–1328, 2004.

[21] R. Silaghi, F. Fondement & A. Strohmeier, "Towards an mda-Oriented uml Profile for Distribution". In *Proceedings of the 8th IEEE Enterprise Distributed Object Computing Conference (EDOC 2004)*, 2004.

[22] D. Regep, Y. Thierry-Mieg & F. Kordon, "Modélisation et vérification de systèmes répartis : une approche intégrée avec LfP". In *Proceedings of AFADL'03*, January 2003.

[23] H. Saiedian, "An Invitation to Formal Methods". *Computer*, 29(4) :16–17, 1996.

[24] A. Singhai, A. Sane & R. H. Campbell, "Quarterware for Middleware". In ICDCS '98: *Proceedings of the The 18th International Conference on Distributed Computing Systems*, page 192,Washington, DC, USA, 1998. IEEE Computer Society.

[25] M. Team. "The MORSE Project home page", http://morse.lip6.fr.

[26] Y. Thierry-Mieg C. Dutheillet & I. Mounier, "Automatic Symmetry Detection in Well-Formed Nets". In Proc. *Of ICATPN 2003*, volume 2679 of *Lecture Notes in Computer Science*, pages 82–101. Springer Verlag, June 2003.

[27] Y. Th.-Mieg, J.-M. Ilié & D. Poitrenaud, "A symbolic Symbolic State Space". In *Proc. of the 24th IFIP WG 6.1 Int. Conf. on Formal Techniques for Networked and Distributed 9 Systems (FORTE'04)*, volume 3235 of *LNCS*, pages 276–291, Madrid, Spain, September 2004. Springer.

[28] N.Wang, K. Parameswaran, D. C. Schmidt & O. Othman, "The design & Performance of Meta-Programming Mechanisms for Object Request Broker Middleware". In *COOTS*, pages 101–118. USENIX, 2001.

Les auteurs

Frédéric Gilliers est docteur de l'Université P. & M. Curie et enseignant à l'Université Paris X. Ses activités se concentrent sur la génération automatique de programmes répartis à partir de spécifications de haut niveau.

Fabrice Kordon est professeur à l'Université P. & M. Curie et responsable du thème Systèmes Répartis et Coopératifs au Laboratoire d'Informatique de Paris 6 (LIP6). Ses centres d'intérêts sont à la convergence des systèmes répartis, des méthodes formelles et du Génie Logiciel. Il a lancé de nombreux projets dans ces domaines (CPN-AMI, PolyORB, MORSE, etc...).

Yann Thierry-Mieg est Maître de Conférences à l'Université P. & M. Curie. Ses activités concernent la modélisation et la vérification formelle de systèmes complexes par model checking. Il se focalise en particulier sur l'exploitation des spécifications UML pour les techniques de vérification.