# Vers l'utilisation d'outils de validation de protocoles dans UML

**Claude Jard — Jean-Marc Jézéquel — François Pennaneac'h**

*IRISA/CNRS*
*Campus de Beaulieu*
*35042 Rennes Cedex*
*Email : (jard,jezequel,pennanea)@irisa.fr*
*Web : www.irisa.fr/pampa/UMLAUT*

RÉSUMÉ. *Malgré un intérêt croissant pour les méthodes formelles et les outils de validation et de vérification associés pour aider au développement d'applications réparties, leur impact réel reste limité. Un des freins principaux réside à notre avis dans le fait qu'elles ne sont pas correctement hébergées dans la chaîne de conception. La construction et la maintenance de systèmes répartis ouverts s'appuient de plus en plus sur un processus de conception objet, et des notations standards comme UML fondées sur cette notion d'objet. Nous proposons de montrer comment un certain nombre d'outils de validation pourraient trouver place dans un cadre de conception («framework»), permettant la validation graduelle et continue d'une application répartie, depuis une analyse exprimée en UML et jusqu'à l'implantation. Nous illustrons notre approche avec le cas d'école du protocole du bit alternant.*

ABSTRACT. *Despite a growing interest in formal methods and related validation and verification tools, the development of distributed systems does not often rely on them. We claim this is mainly due to formal methods lack of support for modern software life cycles. The construction and maintenance of open distributed systems are more and more based on object-oriented software development. We investigate how frameworks may help to embed formal validation techniques in an object-oriented process based on the use of the UML notation. We illustrate our approach with the Alternating Bit Protocol, and we show how it meets the goal of an iterative and continuous validation during the whole software life cycle.*

MOTS-CLÉS : *Modélisation objet, validation et vérification, protocoles, génie logiciel, UML*

KEY WORDS : *Object-oriented modeling, validation and verification, protocols, software engineering, UML*

## 1. Introduction

### 1.1. *Pour un cadre de conception objet orienté protocoles*

Il est assez clair aujourd'hui que la maîtrise de la construction et de l'évolutivité d'un grand système réparti passe par l'adoption d'un processus de conception unifié, permettant de traverser en douceur les différentes phases du cycle de développement. C'est ce qui explique le succès des méthodes de conception dites "objets" (OOAD : "Object-Oriented Analysis and Design") qui permettent la construction incrémentale d'un assemblage d'objets de plus en plus complexes et détaillés.

Cependant l'objectif de conception fiable reste encore difficile à atteindre. La difficulté réside principalement à nos yeux dans le flou des sémantiques formelles associées (notamment au regard de tout ce qui touche à la distribution) et les différentes "ruptures de modèles" que cela introduit dans le processus de raffinement.

Pour relever le défi d'un développement continuellement validé des systèmes répartis ouverts fabriqués à base d'objets, il faut particulièrement soigner :

- la prise en compte correcte du parallélisme et de la répartition dans une approche objets, sans trop nuire aux performances des implantations,
- la disponibilité d'outils de validation dans toutes les phases du cycle de développement.

L'introduction du parallélisme dans un langage orienté objet est une question assez difficile. Il est généralement vu de deux façons, suivant que le parallélisme est fortement ou peu intégré à la sémantique du langage.

Pour les adeptes de la non intégration, les constructions parallèles sont simplement superposées au langage objet : elles sont interprétées dans un monde différent ; on peut dire que la sémantique parallèle et la sémantique objet cohabitent (comme c'est par exemple le cas lorsque l'on utilise CORBA). Le résultat est généralement la perte des facilités objets comme l'héritage et la substitution de types dès que l'on manipule du parallélisme et des synchronisations [MAT 93, MES 93].

La deuxième approche consiste à étendre la sémantique d'un langage objet avec une sémantique du parallélisme. Un exemple significatif est Parallel Eiffel [MEY 93], dans lequel il n'y a plus de distinction entre les objets "actifs" et "passifs". Le prix à payer est la manipulation du parallélisme à un très haut niveau (à l'image du schéma du multi-rendez-vous réparti de Parallel Eiffel), ce qui limite sérieusement son application pour de nombreux problèmes.

Fondé sur l'intuition qu'il faut utiliser toutes les facilités du cadre objet, a fortiori pour maîtriser les complications apportées par le parallélisme, et que par ailleurs la recherche d'un langage universel possédant tous les aspects requis pour la programmation parallèle et communicante est un mythe, nous pensons qu'il faut se placer dans le cadre d'une troisième voie, représentée par la conception puis l'utilisation de "cadres de conception" ("frameworks" en anglais).

Un framework fournit un ensemble intégré de fonctionnalités spécifiques à un domaine, implanté par une collection de classes liées entre elles par de multiples schémas

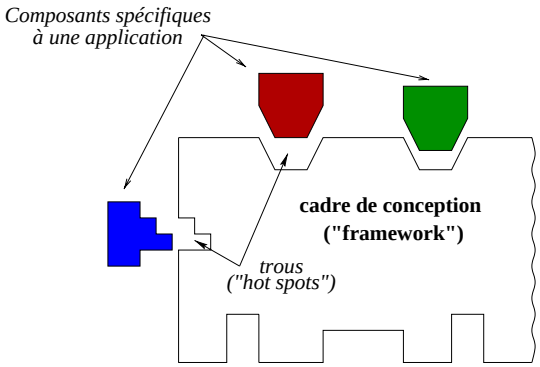


**Figure 1.** *Principe d'un cadre de conception*

(patterns [GAM 95]) de collaboration statiques et dynamiques. Il fournit un modèle d'interaction entre les différents objets instances des classes définies (ou seulement spécifiées pour les classes abstraites) dans le framework. Celui-ci présente en général une inversion du contrôle à l'exécution : alors qu'une application utilisant une bibliothèque s'appuie sur celle-ci, dans le cas d'une application utilisant un framework, c'est le framework qui effectue l'essentiel du travail et appelle ńde temps en tempsż un composant spécifique réalisé par l'implanteur de l'application. Un framework peut donc être vu comme une application semi-complète : c'est-à-dire un squelette [SHA 96] ou schéma de programme "à trous" (figure 1). Des applications complètes sont développées par le remplissage de ces trous ("hotspots" en anglais), ce qui se fait en pratique en héritant et en instanciant des composants paramétrés de frameworks : il suffit donc en quelque sorte d'enficher dans un framework les composants spécifiques de son application pour obtenir une application complète. Dans un contexte de programmation par objets, on s'appuie sur le mécanisme de la liaison dynamique pour dissocier la spécification d'une opération (donnée dans une classe du framework) de son implantation dans une sous-classe, qui fait partie du code applicatif fourni par l'utilisateur du framework.

L'intérêt est que l'on peut concevoir des cadres spécialisés pour un domaine d'application donné. Ces cadres sont destinés à être réutilisés. Du coup, on peut passer du temps à étudier une fois pour toutes la validité du cadre de conception, et surtout l'équiper d'un ensemble d'outils de validation formelle qui serviront à améliorer la qualité du code produit. Bien entendu, dans une approche objets, il faut essayer de rendre les outils de validation utilisables tout au long du développement, depuis les stades de spécifications préliminaires jusqu'au code final.

Il est clair que l'objectif de validation ne peut pas se passer de descriptions formelles. Nous avons besoin de descriptions possédant une sémantique précise permettant de raisonner formellement sur les comportements des programmes. Cet aspect n'était malheureusement pas mis en avant dans les méthodes de conception objets les plus populaires comme Booch [BOO 94] ou OMT [RUM 91] (Object Modeling Technique). Souvent même (généralement en dehors du cadre académique), le flou de la sémantique est présenté comme un argument positif permettant de ne pas trop

contraindre le concepteur. La notation UML («Unified Modeling Language») [RUM 97] tente de combler les lacunes de ses trois prédécesseurs en s'appuyant sur un méta-modèle lui-même décrit en UML et fournissant une base semi-formelle.

### 1.2. *Notre contexte : UML et Eiffel*

Compte tenu de cet état de l'art et pour montrer que l'on peut développer un logiciel réparti dans un cadre objet tout en profitant d'outils de validation tout au long du développement, nous avons choisi d'aborder le problème à la sortie de la phase d'analyse. À ce point, nous pouvons utiliser une notation à la fois suffisamment abstraite et formelle pour décrire précisément la conception, et suffisamment efficace pour aller jusqu'à l'implantation, c'est-à-dire autorisant la génération de code, dans un processus continu de raffinement. Nous utilisons la notation UML comme support de la modélisation, ainsi que le langage Eiffel [MEY 92] pour l'implantation, ces deux approches offrant des concepts communs tels l'héritage multiple, la généricité, la gestion des exceptions et, particulièrement utile à notre approche, l'usage (intégré avec l'héritage) d'assertions pour améliorer la correction des programmes dans l'idée de la «programmation par contrats». Ces nombreuses similitudes entre la notation et le langage facilitent le passage de l'un à l'autre, et des raisonnements similaires pourront s'appliquer aux diverses phases de modélisation avec UML et à l'implantation finale en Eiffel. La sémantique d'UML est définie par le méta-modèle dans [RUM 97], celle d'Eiffel est donnée formellement dans [ATT 93].

Le méta-modèle UML semble apporter une sémantique suffisamment précise pour raisonner formellement sur le comportement des modèles. Une autre caractéristique intéressante de la notation UML est son ouverture : les mécanismes d'extension de la notation intégrés au méta-modèle (annotations, contraintes et stéréotypes) permettent de modifier ou préciser la sémantique des éléments d'un modèle ; il est envisageable d'utiliser ces extensions pour indiquer la distribution, l'asynchronisme des communications, etc. . .

La suite de l'article est structurée en trois parties. Nous commençons par plaider en faveur de techniques de descriptions formelles ouvertes pour permettre une conception/validation incrémentale. Nous illustrons cette idée par le traitement d'un petit protocole de communication, en montrant comment UML peut être utilisé depuis l'analyse jusqu'à une implantation validée. En dernier lieu, nous discutons sur la façon concrète d'intégrer les outils de validation dans ce cycle de développement objet, toujours en nous appuyant sur la notation UML, son méta-modèle et ses possibilités de spécialisation et d'extension.

## 2. Validation de logiciels répartis ouverts fondée sur des descriptions formelles

### 2.1. *Différentes facettes de la validation*

Les outils de validation ont pour objectif de fiabiliser la conception des programmes. Ils essaient d'assurer le concepteur que l'implantation finale fonctionnera correctement sous des hypothèses d'environnement bien identifiées. Ils peuvent être utilisés sur des descriptions formelles à différents niveaux (de la spécification jusqu'à l'implantation) et la qualité des résultats de l'analyse dépend de la complexité du programme à traiter. On peut grossièrement distinguer trois catégories.

- La preuve de propriétés : elle donne une réponse sûre concernant la correction du programme. L'état de l'art sur les modèles du réparti ne permet à l'heure actuelle que de traiter des abstractions simplifiées. Une fois la preuve effectuée, la question délicate consiste à continuer le développement jusqu'à l'implantation tout en préservant les propriétés prouvées.
- La simulation : elle permet d'identifier des comportements erronés dans un environnement contrôlé (généralement centralisé pour les logiciels répartis). La simulation peut traiter des descriptions fines de programmes. La principale difficulté réside dans la définition artificielle de l'environnement d'exécution. Celui-ci est généralement simplifié pour des raisons de coût et peut ne pas être réaliste.
- L'observation et le test : dans cette phase, l'environnement d'exécution est réel. La difficulté ici, amplifiée par le parallélisme asynchrone et réparti, est d'observer les événements pertinents et de contrôler le non-déterminisme inhérent au système.

On sait maintenant que ces techniques sont vraiment complémentaires et que la performance d'un atelier de validation tient dans la possibilité de les utiliser conjointement et de façon continue tout au long des différentes phases de développement.

### 2.2. *Difficultés dans l'utilisation des techniques formelles pour les protocoles*

Pendant ces dix dernières années, le domaine des protocoles a fait l'objet d'une grande attention concernant les techniques de description formelle (TDF) et leur application. Nous allons maintenant examiner les freins existants à leur intégration dans le cycle de développement.

Il est en effet décevant de remarquer que l'impact de la validation fondée sur les TDF normalisées (comme SDL [SDL 87], Estelle [COU 87, ISO 89] ou Lotos [ISO 85]) est encore faible dans l'industrie en regard de l'excellence des résultats obtenus sur un certain nombre de projets pilotes [LAI 97b, JÉZ 90, GAR 97]. Il semble que ceci soit en dernière analyse principalement dû à une mauvaise intégration de ces technologies innovantes dans les méthodes de développement largement utilisées dans l'industrie.

Même en se projetant un peu dans l'avenir, on voit mal comment les TDF actuelles vont pouvoir bénéficier des principes du génie logiciel moderne. Il est de toute façon extrêmement difficile de les imaginer pouvoir aller jusqu'à la définition d'implantations d'applications réparties réelles de grande taille [LAI 97a]. De plus, chacun des

modèles sémantiques qui fondent ces TDF est assez fermé. Prenons comme exemple la sémantique de synchronisation dans SDL et Lotos. Si vous avez choisi les files de communication SDL pour représenter l'interaction entre les couches d'une pile de protocoles, il sera très difficile sans remettre en cause la structure générale du programme de transformer ces files en appel local de fonctions. Inversement, si vous avez choisi le mécanisme de multi-rendez-vous de Lotos, vous aurez beaucoup de mal à le mettre en œuvre efficacement sur les files du système de communication.

Force est de constater que l'utilisation des TDF n'empêche pas l'existence de ruptures de modèles dans le cycle de développement. Non seulement ceci fragilise la transmission des résultats de validation tout au long de la chaîne de développement, mais cela rend aussi extrêmement difficile toute la phase de maintenance du système qui se fait au prix d'une rétro-ingénièrie très coûteuse. Puisque la phase de maintenance de grands systèmes peut représenter jusqu'à trois à quatre fois le coût du développement initial, le problème est inquiétant. Seuls les systèmes hautement critiques justifient actuellement un tel investissement.

### 2.3. *Pour une intégration de la validation dans le cycle de vie objet*

Dans le génie logiciel moderne, les méthodes d'analyse et de conception par objets, alliées à une implantation orientée objet, prêchent pour l'utilisation d'un cadre conceptuel unique utilisable dans tout le cycle de vie. La première étape de la méthode consiste à modéliser le domaine du problème en identifiant les objets et leurs relations. Cette analyse constitue la base stable sur laquelle la phase de conception ("design") va pouvoir se construire. Partant des résultats de l'analyse, la phase de conception permet de passer du domaine d'application vers le domaine de calcul : elle définit la stratégie d'implantation et fait les choix et compromis correspondants. Des classes auxilliaires sont généralement introduites pour prendre en compte des "patrons de conception" [GAM 95], traiter des relations complexes ou des aspects spécifiques de l'implantation. La sortie de la phase de conception fournit un canevas pour l'implantation dans un langage objet, laquelle se présente essentiellement comme une extension de la conception.

Les frontières entre ces trois phases ne sont pas rigides, et le fait de n'avoir pas de rupture de modèle permet de procéder à de multiples va-et-vient entre elles, ce qui est particulièrement utile lorsqu'on utilise des cycles de développement incrémentaux [LEV 97] ou en spirale [BOE 87]. Nous plaidons dans le sens de l'extension de cette souplesse du développement orienté objet à la prise en compte de la validation, pas seulement vue comme une question posée a posteriori (comme le suggère le "V" du cycle de vie), mais comme une activité intégrée dans le processus de développement. La démarche, illustrée en figure 2, propose de partir d'une modélisation UML du problème, qui est transformée par le concepteur en une modélisation de la solution. Ce nouveau modèle (toujours exprimé avec UML) fait apparaitre des informations de déploiement et prend donc en compte la dimension répartie du système. Cependant, le fait que des communications entre objets aient lieu à distance n'est qu'implicite. Pour

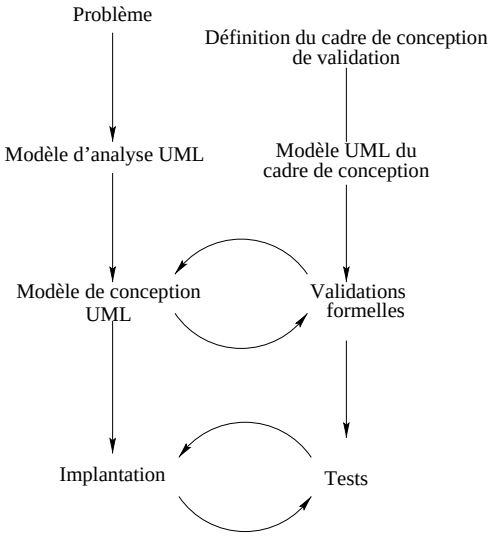


**Figure 2.** *Cycle de vie intégrant la validation formelle*

pouvoir utiliser des outils de validation formelle de ce modèle UML, il faut donc expliciter ces communications, en quelque sorte en intercalant la sémantique du médium de communication. Comme mentionné précédement, ces outils peuvent soit donner des informations sur la validité du modèle de conception, soit générer des tests qui pourront être passés sur une implantation de ce modèle.

Le point clé est de s'appuyer sur des modèles sémantiques bien établis et leurs outils. La notation UML nous semble ici intéressante à plusieurs égards : le méta-modèle fournit la base sémantique nécessaire, cette notation est conçue pour faciliter l'intégration d'abstractions tels les patrons de conception, les composants et leurs collaborations [RUM 97], son modèle dynamique (diagramme des états et d'activité) repose tout comme Estelle et SDL sur les automates à états finis. Cette vision de la validation intégrée va être illustrée dans la suite par un petit exemple de protocole.

## 3. L'exemple du bit alternant

### 3.1. *Introduction au bit alternant*

Ce n'est pas la peine d'aller chercher dans des protocoles à la mode pour illustrer la démarche. Au stade où en est la réflexion, nous avons préféré reprendre la "drosophile" de l'ingénierie des protocoles : le protocole du bit alternant ("Alternating bit protocol") qui a déjà beaucoup servi pour illustrer l'utilisation de descriptions formelles et a permis de nombreuses comparaisons. Notre approche pourra donc être utilement située par rapport aux autres.

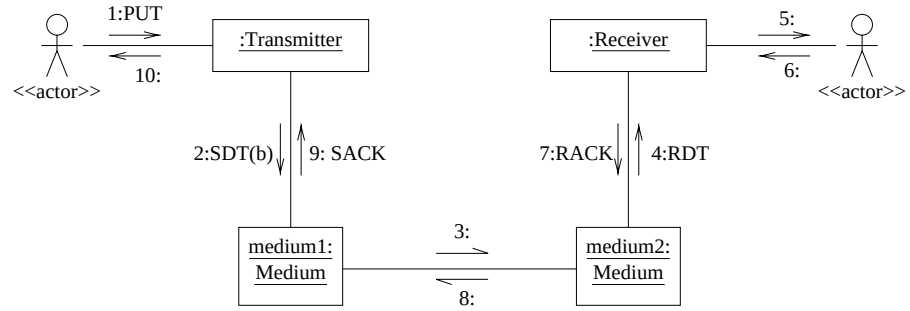


**Figure 3.** *Diagramme de collaboration du bit alternant (modèle dynamique)*

### 3.2. *Analyse UML*

Pour modéliser l'architecture logicielle du protocole, nous avons choisi le cadre objet de la notation UML. Une analyse UML peut être décomposée en quatre grandes «vues» :

- un modèle statique (fondé sur la description de classes et de leurs relations mutuelles) qui décrit la structure statique du système et les relations entre les composants ;

- un modèle dynamique qui décrit le comportement temporel des objets du système. Divers types de diagrammes peuvent être utilisés pour préciser ce modèle : diagrammes de séquences (scenarii faisant intervenir le temps comme une dimension explicite), diagramme de collaborations (voir figure 3), diagrammes d'activités (notation proche de celle utilisée par SDL) ou encore diagrammes de Harel (fondés sur des notions d'états et d'événements) attachés à une classe ;

- des cas d'utilisation qui précisent le fonctionnement du système sur des exemples concrets ;

- un modèle architectural (sous-systèmes, localisation des composants et des objets sur les processeurs d'un système réparti) ;

Ces vues constituent des projections du modèle UML, vu alors comme un objet multi-dimensionnel. Ces projections ne sont pas orthogonales : les vues doivent être cohérentes. Par exemple dans le diagramme de collaboration présenté en figure 3, les différents objets représentés sont des instances de classes apparaissant obligatoirement dans le modèle statique (figure 6). On y distingue un module émetteur (":Transmitter") et un module récepteur (":Receiver") communiquant par l'intermédiaire d'un medium de communication muni de deux points d'accès ("medium1" et "medium2"), un pour chacun des deux modules. La numérotation des échanges de messages précise ici le déroulement séquentiel d'une communication donnée. Sous l'action d'un intervenant extérieur («actor»), l'émetteur envoie un message de données ("SDT") portant un bit de contrôle "b" et reçoit ici un accusé ("SACK") de réception portant aussi un bit de contrôle (un autre diagramme de collaboration pourrait être utilisé pour montrer le cas où il reçoit un signal d'accusé erroné "SACKe", car dans notre modèle les médias de communication peuvent perdre ou corrompre les données et les accusés). De façon sy-

métrique, le récepteur reçoit une donnée (“RDT”), qui ici n’est pas erronée, et renvoie un accusé (“RACK”).

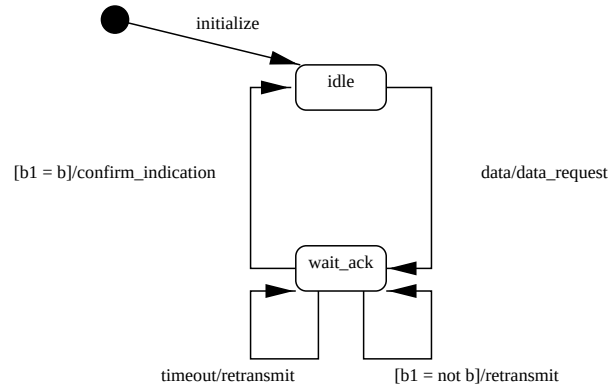


**Figure 4.** *Diagramme d’état de l’émetteur (modèle dynamique UML)*

Dans le modèle dynamique, les scenarii et diagrammes de collaborations ne présentent que des exemples de comportements dynamiques (i.e. des traces d’exécutions attendues). A partir de ces exemples, il est néanmoins possible de préciser la dynamique d’un objet en associant à sa classe un diagramme de Harel, comme illustré en figure 4 pour l’émetteur. Un tel diagramme se présente comme un diagramme états-transitions, les transitions étant déclenchées par la réception d’événements asynchrones (messages, signaux, échéances de temporisateurs, etc.)

### 3.3. *Un cadre de conception adapté*

Nous allons montrer comment mettre en œuvre ce modèle UML dans un cadre de conception (“framework”) adapté pour la validation et l’implantation du protocole.

Il s’agit de transformer le modèle de conception UML en un modèle intégrant les mécanismes nécessaires à sa validation. Pour cela nous introduisons un cadre de conception contenant les fonctionnalités de simulation et d’observation permettant de valider l’implantation du modèle de conception. Si celle-ci ne répond pas aux spécifications, les résultats de la validation servent à la correction du modèle d’analyse, à la génération d’un nouveau modèle de conception et ainsi de suite, jusqu’à l’obtention d’une implantation conforme, comme suggéré par la figure 2.

Nous décrirons en détail la structure et le fonctionnement de ce cadre de conception pour les protocoles.

Un cadre de conception consiste en une collection de classes liées entre elles par de nombreux schémas (patterns) de collaboration statiques et dynamiques. Il fournit un modèle d’interaction entre les différents objets instances des classes définies (ou seulement spécifiées pour les classes abstraites) dans le cadre.

L’architecture générale du cadre de conception retenu pour notre étude est pré-

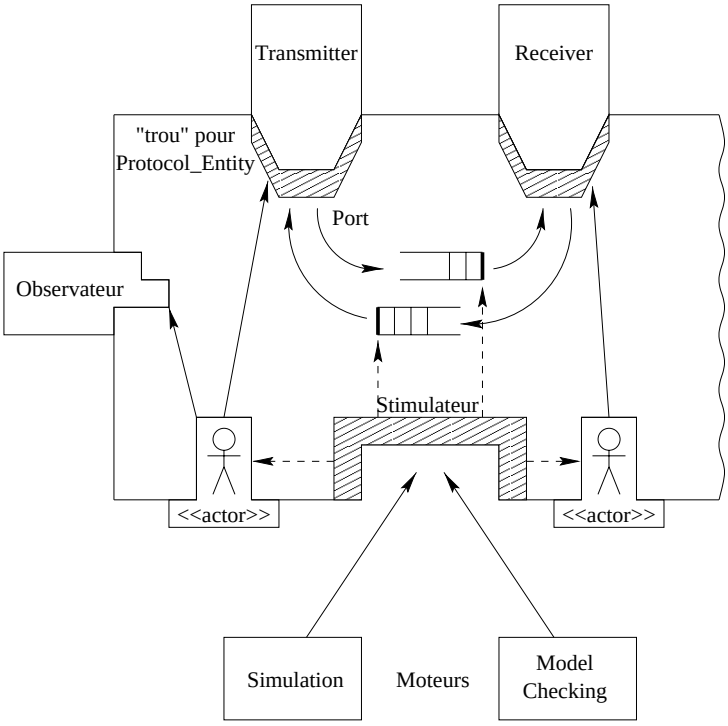


**Figure 5.** *Un cadre de conception pour les protocoles*

sentée figure 5 : elle reprend la figure 1 en insérant dans les “trous” les parties manquantes, appelés greffons (“plug-in”). Notre cadre est accompagné d’un ensemble de greffons permettant de mettre en œuvre les diverses facettes de la validation précédemment décrites (paragraphe 2.1). Parmi ces greffons, figurent sur ce schéma ceux offrant la preuve de propriétés (greffon «model-checking»), la simulation (greffon «simulation»), ou encore l’observation (greffon «observateur»).

Le diagramme de classes UML de la figure 6 présente le modèle statique détaillé du protocole du bit alternant plongé dans notre cadre de conception. Les classes sont figurées par des rectangles (les boîtes blanches correspondent aux classes spécifiques au protocole du bit alternant, alors que les boîtes grises représentent des classes du cadre de conception) et les relations entre classes par des lignes. Les relations multipoints sont repérées par un losange. Un triangle symbolise la relation d’héritage (par exemple Transmitter hérite de ProtocolEntity). Les numéros situés aux extrémités d’une relation indiquent la cardinalité de la relation. L’utilisation de l’italique indique qu’une méthode ou une classe est abstraite.

Les principales abstractions de notre cadre de conception sont :

- Les objets réactifs, héritant de la classe REACTIVE et devant définir la méthode `receive (e: EVENT)` pour réagir aux événements, qui peuvent être soit des mes-

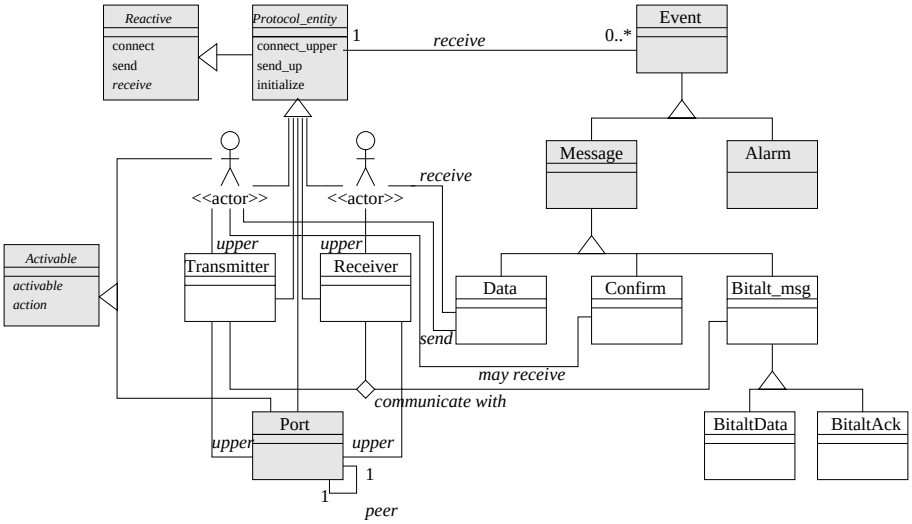


**Figure 6.** *Modèle statique UML du bit alternant*

sages asynchrones ou des signaux, ou encore la notification de l'expiration d'un timer.

- Les objets pro-actifs, héritant de la classe ACTIVABLE, qui doivent définir les méthodes `activable` et `action`. Les objets pro-actifs sont succeptibles d'être exécutés en parallèle, avec une sémantique d'entrelacement entre leurs actions (la méthode `action` étant atomique).

- Les interfaces avec le monde extérieur (modélisées par la classe PORT), qui sont en fait déclinées en plusieurs sous-classes pour prendre en compte les différents plans d'adressage et/ou types de qualité de service (e.g. fiable, non fiable, etc.) disponibles pour le concepteur de l'application.

L'idée centrale est que toute classe interagissant avec un site distant dans un système distribué doit être une sous-classe de REACTIVE, d'ACTIVABLE (ou des deux), et utiliser (éventuellement de manière indirecte) une sous-classe de PORT pour ses communications distantes. Cette interaction avec des objets distants peut être spécifiée par exemple sur un diagramme de déploiement, un diagramme de collaboration (qui permet de caractériser les communications asynchrones), ou encore en utilisant l'un des mécanismes d'extension de la notation UML (stéréotypes, etc.). Dans ce cadre, le modèle dynamique UML est traduisible en méthodes d'un langage objet, soit directement, soit par l'intermédiaire d'un autre modèle dynamique UML obtenu par transformation du modèle de départ :

- l'événement d'activation d'une transition (lorsqu'il existe) est transformé en un paramètre de la méthode associée,

- l'état de départ de la transition, plus les conditions éventuelles sur les paramètres de l'événement ou les variables locales, sont spécifiés par une précondition (en Eiffel, celle-ci consiste en un ensemble d'assertions introduites par le mot-clé require, en UML 1.1 les préconditions sont spécifiées à l'aide de contraintes stéréotypées

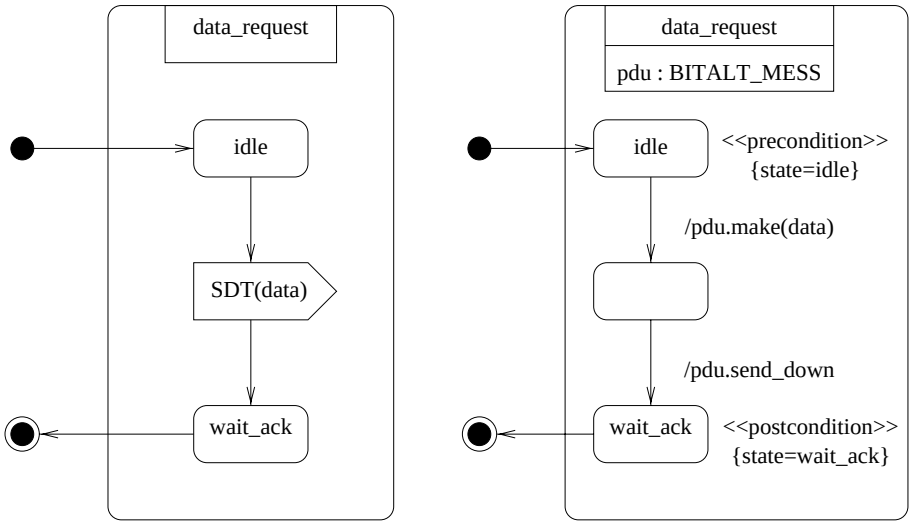


**Figure 7.** *Modélisation UML de la méthode data_request*

«precondition»),

- l'état d'arrivée de la transition est spécifié par une postcondition (les assertions sont introduites en Eiffel par ensure, UML 1.1 utilise la contrainte stéréotypée «postcondition»),

- le corps de la méthode est décrit de façon plus ou moins abstraite en appelant éventuellement des fonctions non encore implantées (introduites par le mot-clé deferred en Eiffel).

La figure 7 présente un exemple de transformation sur le modèle UML en montrant comment le cadre de conception s'intègre à la méthode data_request (simplifiée ici par la suppression du timeout). L'envoi de message à un objet distant, modélisé par la levée d'un signal ("SDT(data)") est transformé de manière à utiliser la méthode send_down de PROTOCOL_ENTITY. La méthode receive, invoquée depuis l'extérieur, constitue le moteur de l'entité de protocole. Elle déclenche les actions en fonction de l'événement en entrée et de l'état interne de la méthode. L'implantation d'une telle méthode repose sur un mécanisme d'aiguillage double ("double dispatch") classique [GUI 95].

## 4. Accueil des outils de validation

On va montrer dans ce paragraphe comment différentes validations peuvent être conduites au cours du raffinement du code obtenu, depuis les premières versions du modèle de conception jusqu'à l'implantation finale.

Puisque notre système a été conçu de manière à pouvoir être compilé en un programme réactif composé de transitions gardées dans chaque objet activable, ces dernières peuvent être activées avec des objectifs variés.

Par exemple, une simulation exhaustive (“Model-checking”) peut être conduite en définissant un moteur qui, partant d’un état initial, va explorer toutes les suites possibles de transitions.

Pour des systèmes plus complexes, une simulation intensive sera plus adaptée (on n’explore que partiellement, mais à la volée, le graphe de comportement). Le moteur de simulation est défini par un objet ordonnanceur disponible une fois pour toute dans la librairie du cadre de conception. Il dispose d’une liste d’objets ACTIVABLE dont il pilote le tir des transitions, par exemple en tirant aléatoirement à chaque pas de simulation une transition (appel de la méthode action sur l’objet activable choisi) parmi toutes celles qui sont tirables (i.e. pour lesquelles l’appel à la méthode activable retourne la valeur *vrai*). Dans notre étude de cas sur le bit alternant, les objets activables sont les deux acteurs modélisant les couches hautes du protocole, les deux ports de communication modélisant le réseau et sa latence, ainsi que les éventuels temporisateurs qui auraient pu être armés.

Il est aussi possible d’observer le système en utilisant un observateur comme ceux de Véda [JAR 88]. Un tel observateur est informé de la suite des actions atomiques (transitions) effectuées par le système. Il est implanté conformément au “Design Pattern” connu sous ce même nom d’Observateur et décrit dans [GAM 95]. Si l’on prend soin de coder les gardes des transitions de l’observateur comme en préconditions des routines les implantant, les erreurs de séquencement du protocole seront détectées comme des violations de préconditions dans l’observateur.

Nous systématisons ici l’idée sous-jacente aux environnements de validation “ouverts” comme CADP (Caesar-Aldébaran-Package) [FER 92] ou Géode.

Un point important est que l’obtention d’une implantation à partir des descriptions validées va se faire naturellement en remplaçant simplement le moteur de simulation par un moteur efficace spécialisé et la classe Port par la connexion effective à la véritable interface réseau.

## 5. Conclusion

L’idée d’utiliser des outils de validation traditionnels pour les logiciels répartis dans un cadre de programmation objet semble à portée, et améliorerait considérablement les environnements de programmation existants. Ceci se fait au prix de la définition de “frameworks” offrant des outils de validation ouverts. Ces outils accueillent les descriptions spécifiques de protocoles obtenues par une discipline de structuration des objets. Bien sûr, cet article n’est qu’un premier pas exploratoire dans cette direction, qui montre seulement la possibilité de l’approche. Cependant, au delà du problème jouet du bit alternant, nous avons appliqué un sous-ensemble de l’idée (validation par simulation intensive) sur un système réel comme l’implantation d’un serveur parallèle haut-débit SMDS avec des résultats de fiabilité intéressants [GUE 94].

Nous essayons à présent de réaliser complètement un tel cadre de conception pour les protocoles. Il s’agit d’automatiser les transformations de modèles UML qui ont été évoquées au paragraphe 2.3, afin d’adapter automatiquement ces modèles à notre

cadre de conception pour les protocoles. Pour cela, un outil permettant de transformer des modèles UML en raisonnant sur un ensemble de règles de transformations définies au niveau du méta-modèle est en cours de réalisation. Le but visé est l'intégration dans un atelier industriel, de manière à favoriser l'application du cycle de développement itératif présenté en figure 2. Un démonstrateur a été réalisé en coopération avec la société Softeam sur la base de l'atelier de génie logiciel *Objecteering* (cf. http://www.irisa.fr/pampa/UMLAUT). Ce démonstrateur a été utilisé pour traiter une étude de cas provenant du domaine du commerce électronique.

## Bibliographie

[ATT 93] ATTALI I., CAROMEL D. et OUDSHOORN M., « A Formal Definition of the Dynamic Semantics of the Eiffel Language ». In *Proc. of the Sixteenth Australian Computer Science Conference (ACSC-16)*. Brisbane, Australia, 1993.

[BOE 87] BOEHM B. W., « Improving Software Productivity ». *Computer*, vol. 20, n° 9, p. 43–57, September 1987.

[BOO 94] BOOCH G., *Object-Oriented Analysis and Design with Applications*. Benjamin Cummings, 2nd édition, 1994.

[COU 87] COURTIAT J.-P., DEMBINSKI P., GROZ R. et JARD C., « Estelle : un langage ISO pour les algorithmes distribués et les protocoles ». *Technique et Science Informatiques*, vol. 6, n° 2, 1987.

[FER 92] FERNANDEZ J.-C., GARAVEL H., MOUNIER L., A. RASSE C. R. et SIFAKIS J., « A Toolbox for the Verification of Programs ». In *International Conference on Software Engineering, ICSE'14, Melbourne, Australia*, p. 246–259, May 1992.

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

[GAR 97] GARAVEL H., GROZ R. et JARD C., Eds., *RAIRO, Technique et science informatiques : Méthodes formelles : validation de systèmes complexes*, vol. 16. Hermes, Juin 1997.

[GUE 94] GUERBER F., JÉZÉQUEL J.-M. et ANDRÉ F., « Conception et implantation d'un serveur SMDS sur architectures modulaires ». Rapport technique 885, IRISA, Novembre 1994.

[GUI 95] GUIDEC F., « *Un cadre conceptuel pour la programmation par objets des architectures parallèles distribuées : application à l'algèbre linéaire* ». Thèse de doctorat, IFSIC / Université de Rennes 1, juin 1995.

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

[ISO 89] ISO, « *Estelle: a Formal Description Technique based on an Extented State Transition Model* ». ISO 9074 TC97/SC21/WG6.1, 1989.

[JÉZ 90] JÉZÉQUEL J.-M., « Experience in Validating Protocol Integration Using Estelle ». In *Proc. of the Third International Conference on Formal Description Techniques, Madrid, Spain*, November 1990.

[JAR 88] JARD C., GROZ R. et MONIN J., « Development of VEDA: a prototyping tool for distributed algorithms ». In *IEEE Trans. on Software Engin.*, vol. 14,3, p. 339–352, March 1988.

[LAI 97a] LAI R., « An Experience in Using EDT to Process the ISO Transaction Processing Estelle Specification ». *J. Systems Software*, vol. 36, p. 137–145, 1997.

[LAI 97b] LAI R., « A Success Story of Verifying a Real Complex Protocol ». *J. Systems Software*, vol. 36, p. 69–83, 1997.

[LEV 97] LEVY N. et SOUQUIÈRES J., « Modeling Specification Construction bu Successive Approximations ». In *6th International AMAST conference*, n° 1349 in Lecture Notes in Computer Science, p. 351–364, Sydney, Australia, December 1997. Springer Verlag.

[MAT 93] MATSUOKA S. et YONEZAWA A., Analysis of Inheritance Anomaly in Object-Oriented Concurrent Programming Languages. In AGHA G., WEGNER P. et YONEZAWA A., Eds., *Research Directions in Concurrent Object Oriented Programming*. MIT Press, 1993.

[MES 93] MESEGUER J., « Solving the Inheritance Anomaly in Concurrent Object-Oriented Programming ». In NIERSTRASZ O., Ed., *Proceedings ECOOP'93*, LNCS 707, p. 220–246, Kaiserslautern, Germany, July 1993. Springer-Verlag.

[MEY 92] MEYER B., *Eiffel: The Language*. Prentice-Hall, 1992.

[MEY 93] MEYER B., « Systematic Concurrent Object-Oriented Programming ». *Communications of the ACM*, vol. 36, n° 9, September 1993.

[RUM 91] RUMBAUGH J., BLAHA M., PREMERLANI W., EDDY F. et LORENSEN W., *Object-Oriented Modeling and Design*. Prentice Hall, New Jersey, 1991.

[RUM 97] RUMBAUGH J., JACOBSON I. et BOOCH G., *Unified Modeling Language Reference Manual*. Addison-Wesley, 1997.

[SDL 87] SDL87, « Special issue on the CCITT language SDL ». *Computer Networks and ISDN Systems*, vol. 13, n° 2, p. 65–134, 1987.

[SHA 96] SHAW M. et GARLAN D., *Software Architecture, perspectives on an emerging discipline*. Prentice Hall, 1996.