Modélisation et vérification de systèmes répartis : une approche intégrée avec **L*f*P**

D. Regep, LIP6-SRC[‡]
Y. Thierry-Mieg, LIP6-SRC[‡]
F. Gilliers, SAGEM SA[*]
F. Kordon[†], LIP6-SRC[‡]

8 juillet 2004

Résumé

L*f*P est un langage de spécification possédant les capacités d'un langage de description d'architecture (ADL). Nous l'utilisons dans une approche de *prototypage par raffinements* dédié au domaine des systèmes répartis. Cette approche utilise un modèle servant de base à la génération automatique de spécifications formelles et de squelettes de programmes répartis. L'objectif est d'assurer l'absence de biais entre le modèle **L*f*P** d'une part, les programmes générés et les propriétés vérifiées sur les spécifications formelles d'autre part. Ainsi, les squelettes de programmes répartis générés automatiquement sont conformes au modèle vérifié.

Dans cet article, les principales caractéristiques de **L*f*P** sont illustrées au travers d'un exemple classique : la station service. Cet exemple sert par la suite à montrer comment les différents éléments **L*f*P** peuvent être traduits en réseaux de Petri. Une dernière section discute des résultats d'analyse de la spécification formelle ainsi obtenue.

Mots-clefs. Approches mixtes, MDA, Méthodes et Approches formelles, Prototypage, Vérification formelle, Réseaux de Petri, Systèmes répartis

1 Introduction

Les systèmes répartis sont délicats à concevoir et à réaliser. Une solution communément acceptée est d'utiliser des intergiciels (middlewares) qui offrent une aide au développement à travers un modèle de répartition tel celui de CORBA [20]. Cependant, ces intergiciels sont difficiles à utiliser dans des cadres contraints tels ceux des applications embarquées réparties. De plus, leur développement doit répondre à des normes strictes nécessitant la validation de toutes les composantes de l'application. Il faut donc valider l'intergiciel pour chaque utilisation particulière ou changement d'architecture. Chaque validation doit inclure la totalité du code qui sera embarqué et les fonctions inutilisées de l'intergiciel (avec le code correspondant) doivent être clairement identifiées. Dans les cas les plus restrictifs, la présence de code mort n'est même pas tolérée et il devient nécessaire d'adapter finement l'intergiciel à l'application considérée, ce qui est en contradiction avec la démarche classique des intergiciels qui visent un haut niveau de généricité. Enfin, le niveau de performances à atteindre pour les applications réparties embarquées est souvent incompatible avec le surcoût induit par les intergiciels classiques.

Dans ce contexte, des approches orientées modèles, telles que MDA (Model Driven Architecture), prônée par l'OMG [22], deviennent intéressantes. Ces approches de développement sont souvent appelées "prototypage par raffinements" (*evolutionary prototyping*) [14]. Le principe est

[*]21 Avenue du Gros Chêne, 95610 Eragny, BP51, 95612 Cergy Pontoise cedex, France
[†]Contact author : Fabrice.Kordon@lip6.fr
[‡]Université P. & M. Curie, 8 rue du Capitaine Scott, 75015, Paris, France.

de concevoir le système en utilisant un modèle exprimé dans un langage dédié au domaine d'application visé (ici, les systèmes répartis). Ce modèle sert ensuite à la génération automatique de programmes conformes à la sémantique du comportement spécifié. Il sert également de base à la vérification de propriétés qui doivent être respectées.

Nous utilisons un langage de spécification, **L*f*P** (Language for Prototyping) [24] pour décrire un système réparti. Nos travaux promeuvent une approche orientée modèle permettant la vérification formelle du système à développer et la synthèse automatique du squelette de contrôle réparti de l'application.

Cet article présente notre proposition de prototypage par raffinements et illustre ses points forts en matière de vérification formelle. **L*f*P** possède des caractéristiques d'un langage de description d'architecture (ADL - *Architecture Description Language*). Les ADL permettent de caractériser une application en termes de composants (un programme s'exécutant sur un nœud) reliés par des connecteurs (communications entre les composants) et soumis à des contraintes (signature d'une méthode, enchaînement de méthodes, etc.). Ils sont utilisés pour décrire des architectures logicielles de manière formelle ou semi-formelle [18] . Par rapport aux langages de programmation, les ADL permettent de structurer la réflexion et de gagner en pouvoir d'abstraction. Ils s'appuient sur la séparation entre la description interne des composants, celle de leurs interfaces, et celle de leur inter-connexions.

L*f*P répond aux besoins des applications réparties. L'objectif est de capturer l'essence du contrôle réparti et de proposer une correspondance sur des architectures variées comme CORBA, Java/RMI [19], Ada/Distributed System Annex [11], ou des bibliothèques de communication. Pour permettre une telle variété, nous nous sommes focalisés sur la définition des relations entre l'application, l'exécutif (les bibliothèques qui assurent la sémantique opérationnelle de **L*f*P**) et l'environnement d'exécution (système d'exploitation et architecture matérielle).

Après un rappel des problèmes actuellement posés par la prise en compte des aspects comportementaux et architecturaux dans les systèmes répartis (section 2), nous présentons notre démarche méthodologique (section 3). Nous décrivons brièvement le formalisme **L*f*P** (section 4), l'utilisons sur un exemple classique (la station service) pour vérifier une propriété de cohérence (section 5).

2 Aspects comportementaux et architecturaux d'un système réparti

UML est le standard de facto pour décrire des applications objets. Cependant, malgré l'existence de nombreux environnements de qualité et d'un soutien industriel indéfectible, UML souffre de lacunes rendant délicate son utilisation dans le contexte des systèmes répartis et/ou embarqués :

- UML [21] est un langage semi-formel approprié pour les aspects structurels d'un système mais il ne permet pas d'en présenter les aspects dynamiques qui restent informellement décrits ;
- Il est reconnu qu'UML ne permet pas de décrire une architecture logicielle au sens ADL du terme [18].

2.1 Formalisation des aspects dynamiques en UML

La sémantique d'UML décrit parfaitement la structure d'un système. Ce n'est hélas pas le cas des aspects dynamiques (ou comportementaux) qui sont exprimés au moyens de différents diagrammes (séquence, collaboration, activité, états) sans que la cohérence entre eux puisse être assurée. Récemment, des groupes tels que pUML [7] ont proposé une sémantique rigoureuse pour la version 2.0 d'UML (à l'aide du langage OCL) mais ces extensions n'abordent toujours pas les aspects dynamiques.

Les aspects dynamiques concernent non seulement les diagrammes décrivant le comportement mais également leurs relations. Dans ce sens, des travaux récents basés sur la théorie des automates d'états, comme les réseaux de Petri Objets/Hiérarchiques [6, 25], ou sur Promela [13] ont prouvé qu'il était possible d'assembler de manière cohérente l'information répartie dans les différents diagrammes de comportement d'UML à des fins de vérification formelle, notamment par model checking [23]. Ces travaux mettent en avant un procédé de synthèse automatique pour les spécifications formelles.

L'utilisation de notations de haut niveau (par exemple, orientées objets) présente l'avantage de fournir des mécanismes de composition et de structuration plus simples à manipuler et fournit une base intéressante pour la traçabilité des informations dans le système. D'autres notations, basées sur l'algèbre des processus (E-LOTOS [4]) ou sur des langages d'assertions (Object-Z [2]), sont également utilisées pour détecter des incohérences et des ambiguïtés dans les modèles UML au début du processus de développement.

Bien que la formalisation d'UML permette l'utilisation de méthodes de vérification formelles, les outils correspondants requièrent une expérience importante, considérée comme prohibitive pour la plupart des projets industriels. Il est dans ce cas souhaitable de cacher la complexité d'utilisation de ces outils aux développeurs en l'encapsulant dans les services rendus par un AGL [15].

2.2 UML et la description d'architectures logicielles

UML n'est pas réellement adapté à la description d'architectures [18]. Pour pallier cela, on peut utiliser des profils afin de lui donner les caractéristiques d'un ADL [16] mais une telle démarche limite la réutilisation qui est son point fort. C'est pourquoi notre approche est inspirée du framework ODP [12], qui fournit un standard ouvert dédié au développement d'applications réparties, tout en étant compatible avec la notation UML.

Pour permettre le développement de logiciels *certifiables* (c'est-à-dire conformes à une norme de référence pour un domaine d'application donné), l'ADL utilisé doit décrire non seulement la structure et la sémantique de l'application, mais aussi les caractéristiques de son environnement d'exécution : protocoles de communication, gestion de ressources, et algorithmes d'ordonnancement. Néanmoins, construire une application dédiée à un exécutif cible n'est pas recommandé car cela limite la portabilité de la solution proposée. Nous résolvons ce problème en intégrant le système dans des *composants externes* définissant des gabarits de comportement ce qui permet de construire l'application de manière plus générique.

Ainsi en séparant clairement l'application de son environnement d'exécution, nous facilitons son déploiement sur des plate-formes différentes, et nous favorisons la réutilisation des composants existants. Nous intégrons de la sorte l'exécutif à la spécification du système sous la forme d'éléments de configuration. Si les paramètres de cette configuration sont bien maîtrisés, les pertes en performances sont négligeables [26]. Néanmoins, chaque instanciation de l'application sur un runtime cible doit donner lieu à la certification du runtime et de l'application. Cette solution reste un bon compromis entre certification et coût de développement.

3 Méthodologie d'utilisation de L*f*P

Notre méthodologie de développement par prototypage est adaptée aux systèmes répartis embarquables. Notre objectif est de lier fortement la spécification du système, la preuve des propriétés énoncées et les programmes correspondants. Cette approche s'appuie sur un *modèle pivot* permettant la vérification formelle (au moyen de réseaux de Petri Colorés) et la génération automatique de programmes réalisant le contrôle réparti du système. Cette démarche permet de travailler au niveau du modèle puisque les programmes sont obtenus automatiquement à un coût faible. Surtout,

ce modèle sert également comme base pour la vérification formelle.

La partie contrôle des systèmes répartis (synchronisation entre les composants, respect de l'intégrité des ressources, etc.) est la plus délicate à réaliser. Elle doit bénéficier d'une attention particulière lors de la modélisation et de la vérification. La génération automatique de programmes permet à un non-spécialiste d'obtenir, à partir du modèle pivot, le squelette de contrôle de l'application conforme à la spécification vérifiée. Ce squelette est ensuite enrichi par du code séquentiel[1] puis déployé dans un environnement d'exécution donné.

L*f*P, le formalisme proposé pour le modèle pivot, peut être vu comme un diagramme complémentaire à UML regroupant les informations suivantes :

- Les données issues des différents diagrammes de comportement (classe, collaboration, séquence, états) sont utilisées pour construire l'ossature d'une spécification **L*f*P**.
- L'usager intègre ensuite les propriétés attendues dans le système (sous la forme d'invariants, de formules de logique temporelle, etc.).
- L'usager ajoute enfin des directives d'implémentation (langage cible, référence à des composants logiciels préexistants, morceaux de code séquentiel, etc.) et des «directives de déploiement» (références à des éléments de l'architecture cible d'exécution) qui seront utilisées par des générateurs de programmes.

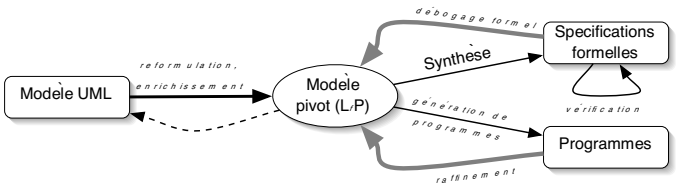

FIG. 1 – Vue générale de la méthodologie.

La figure 1 illustre notre méthodologie. Un diagramme **L*f*P** est construit par raffinement et enrichissement d'une spécification UML. À partir des propriétés identifiées dans le modèle, un processus de synthèse produit des spécifications formelles sur lesquelles sont appliquées des preuves. La sémantique des propriétés à vérifier est utilisée pour réduire la taille du modèle ainsi généré en supprimant les éléments de spécification n'intervenant pas. On est donc amené à générer autant de modèles que de propriétés à vérifier. Par raffinements successifs, on effectue ainsi le déboggage de la spécification au niveau du modèle pivot jusqu'à ce que ce dernier vérifie toutes les propriétés énoncés.

Une fois les propriétés d'un diagramme **L*f*P** démontrées, on obtient une implémentation conforme du système par génération de code. Le diagramme **L*f*P** peut à nouveau être modifié en fonction des observations effectuées lors des exécutions ou en réponse à une évolution du cahier des charges. On obtient alors un cycle de production par raffinements de la spécification **L*f*P** qui comprend les étapes suivantes : synthèse, vérification, correction, génération de programmes, exécution, optimisation, et évolutions.

4 Le formalisme L*f*P

L*f*P est un langage formellement défini permettant de décrire le contrôle d'une application répartie. Il possède à la fois les caractéristiques d'un ADL et celles d'un langage de coordination :

[1] Cette enrichissement n'a de sens que si la technique utilisée préserve la preuve. Ainsi, le code rajouté doit respecter des contraintes : être séquentiel, n'utiliser que les ressources réservées au niveau du squelette de contrôle, etc.

– Il reste lié à une approche basée sur UML. Les concepteurs d'un système utilisent UML pour la phase de conception du système puis passent à **L*f*P** dès qu'il faut exprimer précisément l'architecture logicielle et le comportement des composants répartis.
– Il s'inspire de l'approche RM-ODP [12]. Nous nous sommes en particulier intéressés aux vues *ingénierie, traitement* et *technologie*. ODP nous apporte une démarche d'identification des caractéristiques d'une application particulièrement adaptée aux systèmes répartis.
– Il est défini formellement en se basant sur les réseaux de Petri [8]. Cela rend possible la vérification formelle d'un système dès les premières phases de réalisation, ce qui est dans la ligne de l'approche *Model Driven Architecture* (MDA), prônée par l'OMG [22].

Pour remplir ces objectifs, **L*f*P** définit une notation non ambiguë adaptée aux systèmes répartis basée sur trois vues orthogonales : la vue *fonctionnelle*, la vue *propriété* et la vue *implémentation*.

La vue fonctionnelle décrit l'architecture et le comportement du système. Elle contient :
– une partie déclarative contenant des informations de traçabilité (par exemple, les références vers des composants du modèle UML d'origine) et la déclaration des types et constantes utilisés dans le modèle.
– un graphe hiérarchique décrivant l'architecture du système, ce graphe est composé de *classes*, *média* et *binders*.

Une classe **L*f*P** correspond à une classe instanciable UML. Les média permettent de décrire les schémas de communication (protocoles) entre classes. Les relations d'association, d'agrégation, et de composition d'UML peuvent être capturées au moyen de média. Les binders représentent des points d'entrée (ports de communication) entre les classes et les média.

Les binders **L*f*P** sont des ports de communication (comme on en trouve dans d'autres ADL) et sont placés entre classes et médias. Ils décrivent des caractéristiques simples de la sémantique d'interaction : direction des communications, (a)synchronisme, ordonnancement (FIFO, LIFO), capacités, cardinalités. La possibilité d'associer plusieurs binders à une classe permet de spécifier des protocoles d'interaction complexes, à la différence de l'unique point d'entrée FIFO du modèle événementiel des statecharts UML.

La vue propriété explicite les propriétés à vérifier sur le système. Ces propriétés peuvent être considérées comme des obligations de preuve (au sens des assertions dans B[1]). Ces propriétés sont exprimées en complément de la vue fonctionnelle. Elles prennent la forme d'invariants (pour exprimer une exclusion mutuelle ou une condition sur un ensemble de variables), de formules de logique temporelle (pour exprimer une causalité entre des événements), ou de post-conditions sur les actions. Les informations de cette vue sont exploitées pour la vérification formelle ainsi que pour l'insertion de runtime-checks dans les programmes générés.

La vue implémentation définit les contraintes d'implémentation du système comme l'environnement d'exécution ciblé, le langage utilisé par le générateur de code, ou encore des informations de déploiement du système. Nous nous sommes inspirés de la notion de tag dans UML, ces derniers permettent d'associer toute sorte d'informations ponctuelles aux entités de **L*f*P**. Les informations sont exploitées pendant la génération automatique de programmes.

4.1 Un exemple : la station service

Cette section présente un exemple simple, la station service introduite dans [10], destiné à illustrer les principales caractéristiques de **L*f*P**. Cet exemple a été souvent réutilisé pour comparer l'efficacité de techniques de vérification, comme dans [5, 27].

Considérons le comportement simplifié d'une station service dont le diagramme de classe UML est donné en figure 2 . Le système est composé de quatre entités : l'opérateur (`operator`), les clients (`client`), les pompes (`pump`) et le système d'information de la station (`infosystem`). Lorsque le client entre dans la station, il prépaye l'opérateur, et reçoit un *ticket* qui comporte son

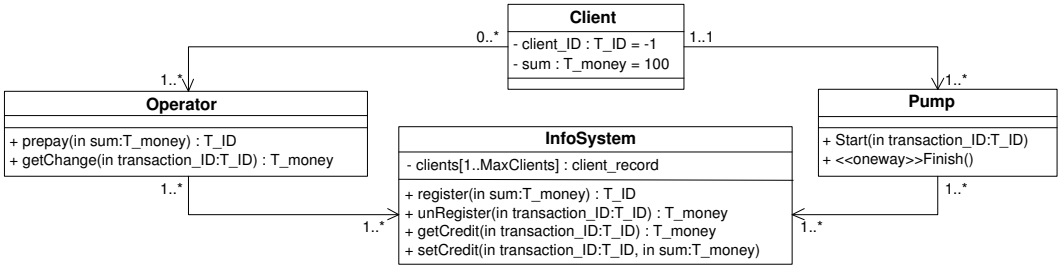

FIG. 2 – Le diagramme de classe UML de la station service.

identifiant. Le client se rend ensuite à une pompe, insère son *ticket* et remplit son réservoir en étant limité par la somme d'argent prépayée. Enfin il retourne voir l'opérateur pour récupérer sa monnaie avant de quitter la station.

Les informations concernant les clients sont centralisées dans le système d'information de la station (**`infosystem`**). L'opérateur enregistre le client (opération **register**) lorsque celui-ci effectue son prépaiement (**prepay**). La pompe récupère le plafond (**getCredit**) attribué à ce client quand il fait **start** avec son ticket, et met à jour le crédit du client (**setCredit**) lorsque le message **finish** est envoyé par le client. L'opérateur retire le client du système d'information (**unRegister**) lorsque celui-ci vient récupérer sa monnaie (**getChange**).

4.2 Le diagramme d'architecture de la station service

Les classes (représentées par des rectangles) du schéma d'architecture d'un modèle **L*f*P** sont complétées par des média (les tubes) décrivant les protocoles de communication entre les composants du système, correspondant aux associations UML.

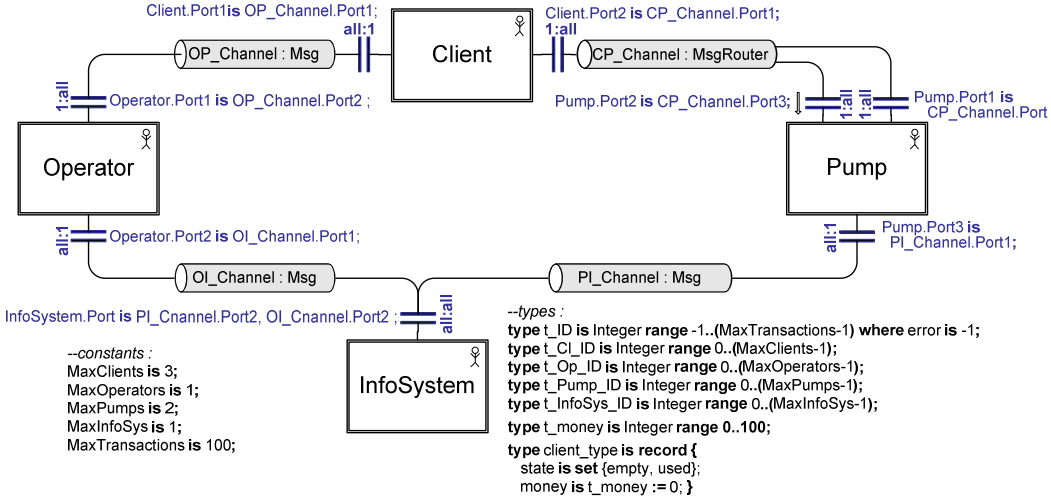

FIG. 3 – Le diagramme d'architecture logicielle **L*f*P** de la station service.

Les média sont connectés aux classes via des *binders* (double barre noire) correspondants à des points d'entrée ou de sortie de messages. Le média décrit le comportement d'un lien de communication (est-il fiable ? préserve-t-il l'ordre des messages ?). Le binder modélise l'interaction entre un média et la classe : il permet d'indiquer s'il y a une file d'attente par instance de classe (cardinalité *1* du coté de la classe) ou une file d'attente unique pour l'ensemble des instances de classe (cardinalité *all*). Un *binder* peut également contenir des informations comme la capacité maxi-

male de la file d'attente correspondante ou le sens de circulation des messages (unidirectionnel, bidirectionnel).

Le diagramme d'architecture **L*f*P** de ce système est présenté en figure 3. Quatre classes **L*f*P** correspondent aux quatre classes UML. Une classe **L*f*P** dispose d'un flot d'exécution propre exprimé au moyen d'un automate. Le concept est clairement inspiré d'un `task type` en Ada [11] : chaque instance de classe dispose de son propre compteur ordinal et correspond à un processus ou une thread.

Dans notre exemple, les classes communiquent à travers des canaux bidirectionnels ; les média auront donc un schéma unique décrivant les caractéristiques de ce type de communication. La seule exception est le média connectant les clients aux pompes CP_channel : selon la nature du message que le client adresse à la pompe, il sera aiguillé sur l'un des deux ports (binders) de la pompe.

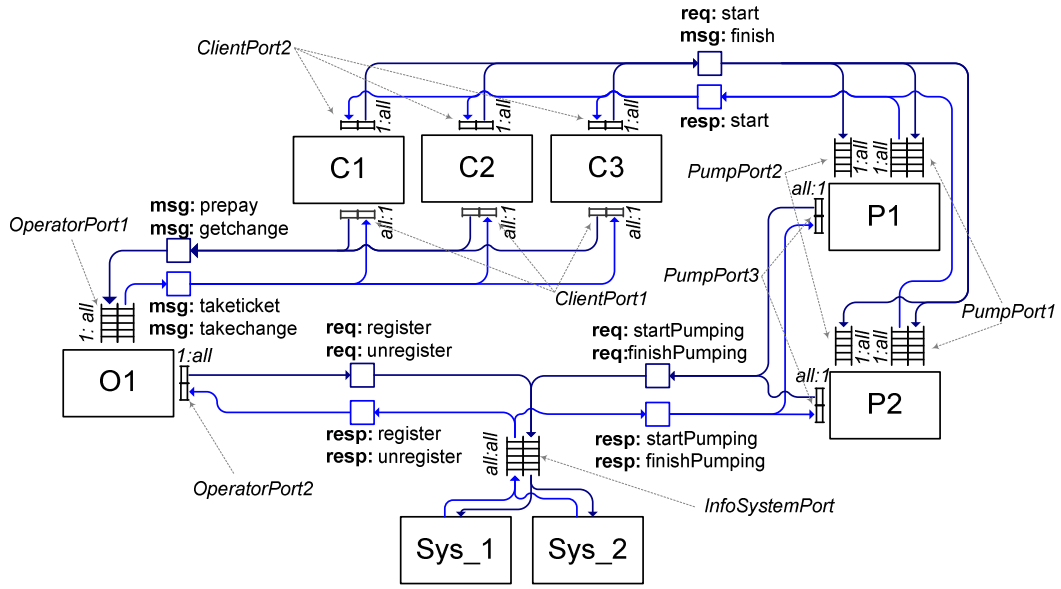

FIG. 4 – Diagramme d'architecture déplié, pour 3 clients, 2 pompes, 2 InfoSystem et un opérateur

La figure 4 présente le dépliage du diagramme d'architecture pour trois clients, deux pompes, un opérateur et deux systèmes d'information. Les classes et média **L*f*P** seront détaillés plus tard, nous précisons ici la signification des cardinalités portées par les binders à travers un diagramme de déploiement **L*f*P**. Les binders de cardinalité *1-ALL* se traduisent par deux files de messages (entrants et sortants) pour chaque instance de classe connectée à un média commun partagé par tous. Le binder *ALL-ALL* qui lie les systèmes d'information au reste du système correspond à une file unique partagée par les deux instances *I1* et *I2* et commune aux médias OI_channel et PI_channel représentés sur la figure 3. Le fonctionnement de cette file est similaire à celui d'une file d'attente : dès qu'un guichet se libère (**`infosystem`**) le premier client de la file est servi (requête d'un opérateur ou d'une pompe). Les communications d'une pompe sont réalisées à travers deux binders bi-directionnels et un binder unidirectionnel.

4.3 Description d'une classe

Le diagramme de comportement (**L*f*P**-BD ou *Behavioral Diagram*) est une représentation graphique hiérarchique du comportement d'une classe. Il exhibe les actions que peut exécuter une instance de classe ou de média en fonction de son état interne. Le **L*f*P**-BD permet ainsi de spécifier la façon de se servir d'une entité **L*f*P**, contraignant ainsi l'ordre des appels aux services.

On définit de la sorte le comportement interne d'une classe **L*f*P** mais aussi le protocole (i.e. enchaînement de méthodes) qui doit être respecté pour en garantir un bon fonctionnement. Cette notion augmente la réutilisabilité des composants définis en **L*f*P** car on peut vérifier qu'un nouveau composant s'intègre correctement dans un système pré-existant (i.e. le protocole qu'il offre est compatible). En contraignant les possibilités d'utilisation du composant, on prévient son usage abusif par l'environnement. On s'assure donc à travers cette *signature comportementale* que, quel que soit le comportement de l'environnement, le composant préserve sa cohérence interne.

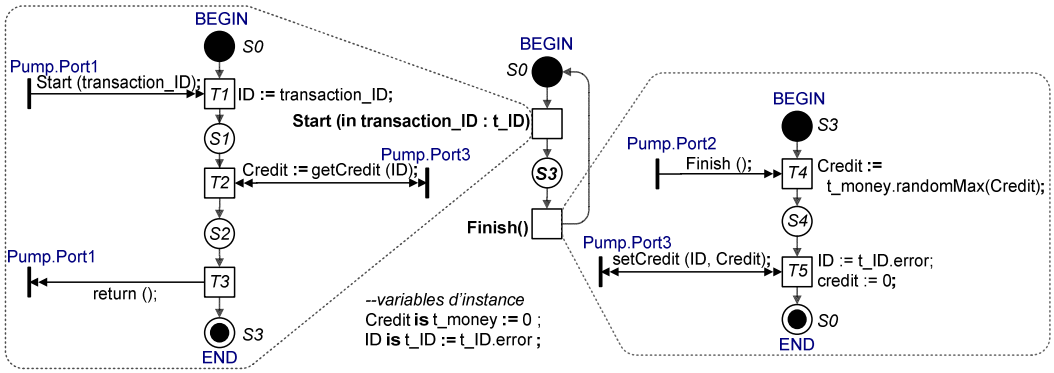

FIG. 5 – Le diagramme de comportement (**L*f*P**-BD) déployé de l'objet **`pump`**.

La figure 5 exprime le comportement de la pompe à travers son **L*f*P**-BD. Au centre le **L*f*P**-BD principal (main-**L*f*P**-BD) de la classe décrit l'ordre dans lequel les appels aux services offerts par le composant doivent s'effectuer. Il exprime que la méthode *start* doit précéder la méthode *finish*. De part et d'autre, les **L*f*P**-BD des deux méthodes, sont déployés avec les instructions et les communications associées.

- Les **déclarations** de types et de variables prennent la forme d'annotations sur le diagramme. La visibilité des déclarations est limitée au diagramme contenant la définition ainsi qu'aux diagrammes qu'il contient hiérarchiquement. Le contexte d'une classe **L*f*P** contient des variables locales (une copie par instance de classe), des variables partagées entre les instances d'une classe (notées «static»). Toute variable **L*f*P** doit être initialisée, mais on peut définir une valeur par défaut à la déclaration du type de la variable.
  La pompe dispose de deux variables locales, **ID** et **Credit**, qui permettent respectivement de mémoriser l'identité du client en cours de traitement, et le plafond qui lui est associé.
- Les **états** représentent les étapes de l'exécution du comportement d'une classe. Deux états spéciaux sont distingués : `BEGIN` et `END` qui correspondent respectivement à l'état initial et à l'état final de l'exécution. Les **L*f*P**-BD n'ont qu'un seul état initial. Pour une méthode l'état final correspond à l'état suivant du main-**L*f*P**-BD, l'état final du main-**L*f*P**-BD correspond à la destruction de l'instance.
  Ainsi l'état initial `S0` est le premier état de la méthode *start*, et l'état `S3` intermédiaire est à la fois l'état final de *Start* et l'état initial de *Finish*.
- Les **transitions** représentent les actions à effectuer en fonction de l'état courant d'une instance de la classe. Une transition peut référencer un rôle ou une méthode d'une classe, donc être décrite par un sous-**L*f*P**-BD (hiérarchique). Une garde spécifie les préconditions à satisfaire pour qu'elle puisse être tirée. Le début de la hiérarchie est prédéfini : les transitions du main-**L*f*P**-BD référencent soit des descriptions de méthodes, soit la description de rôles, qui référencent eux-même la description de méthodes. Ces différents niveaux de hiérarchie sont définis pour offrir différentes vues sur le comportement d'un système, selon la granularité de description souhaitée.

- On peut associer aux transitions un **invariant** ou **post-condition** utilisé pour la vérification ou pour définir un runtime check à insérer lors de la génération du squelette du programme.
- On associe du **pseudo-code séquentiel** aux transitions. Ce pseudo-code peut modifier les variables accessibles à ce niveau hiérarchique au moyen d'un jeu d'instructions prédéfini spécifique à chaque type manipulé (lecture et écriture des variables simples, accès indexé aux tableaux, comparaisons, additions ...). **L*f*P** dispose de types prédéfinis ainsi que d'opérateurs associés (types énumérés et numériques discrets, tableaux, multi-ensembles...). Les utilisateurs peuvent également décrire leurs propres types par composition des types prédéfinis ainsi que les opérations associées. Les types ainsi définis doivent être discrets et les opérations non ambiguës (e.g. s'exprimer sous la forme $f(v_1,...,v_n) = valeur$). On fournit de plus quelques structures de contrôle courantes : itérateurs et conditionnelles.
  Par exemple, la transition `T4` (figure 5) fait appel à une opération prédéfinie RandomMax, qui permet d'obtenir une valeur aléatoire sur un domaine majoré par l'argument. Cette modélisation permet de prendre en compte toutes les valuations possibles de dépense d'essence, et correspond lors de la génération de programme à un appel à une primitive de service de la pompe (module *externe*), liée au programme. Il faut s'assurer que cette primitive a bien ce comportement pour que le modèle soit fidèle à la réalité.
- **L*f*P** propose des **sémaphores** et des **barrières de synchronisation**. Les sémaphores binaires protègent des sections critiques ; les barrières ont le comportement défini dans MPI [17] (multi-rendez-vous).
- On dispose également de **constructeurs** de classes **L*f*P**, qui créent une nouvelle instance de classe (dans l'état BEGIN du main-**L*f*P**-BD), et initialisent son contexte. Le nombre d'instances maximum présent dans le système doit cependant être borné pour conserver le caractère fini du système. Cette borne peut être définie dans la vue implémentation. Si tel n'est pas le cas, l'approche associe à ce constructeur une obligation de preuve afin de s'assurer qu'il existe bien une borde structurelle finie correspondant au nombre maximum de processus créés.

4.4 Description d'un média

Les médias décrivent les protocoles de communication entre les classes du système. Il est possible de les utiliser comme élément de base, ou de les composer pour former des schémas de communication plus complexes. Les médias sont déduits des associations, agrégations, ou compositions UML. Ils sont déclarés de façon générique, mais leur instanciation est paramétrée par les classes frontières.

Un média consiste en :

- une partie déclarative similaire à celle d'une classe. Elle spécifie les éventuels types et variables locales au média.
- un **L*f*P**-BD qui exprime le protocole de communication. Moins complexe que le contrat comportemental d'une classe, le **L*f*P**-BD d'un média ne comporte pas de méthodes. Un média peut être avec ou sans mémoire, dans ce dernier cas il ne comporte ni états ni variables persistantes.
- des ports d'interaction spécifiant les binders **L*f*P** auxquels un média peut se connecter. Ces ports sont similaires aux *bindings points* ODP.

Un message **L*f*P**est composé d'une en-tête de taille variable et d'une partie qui véhicule les données. La structure de l'en-tête est définie dans le média et est utilisée pour le routage des messages. Elle est composée de plusieurs champs implicites (source, destination, et nom du message) et d'autres paramètres optionnels à définir par l'utilisateur.

La figure 6.a montre le patron générique du média MsgRouter qui relie les **`client`** aux **`pump`**.

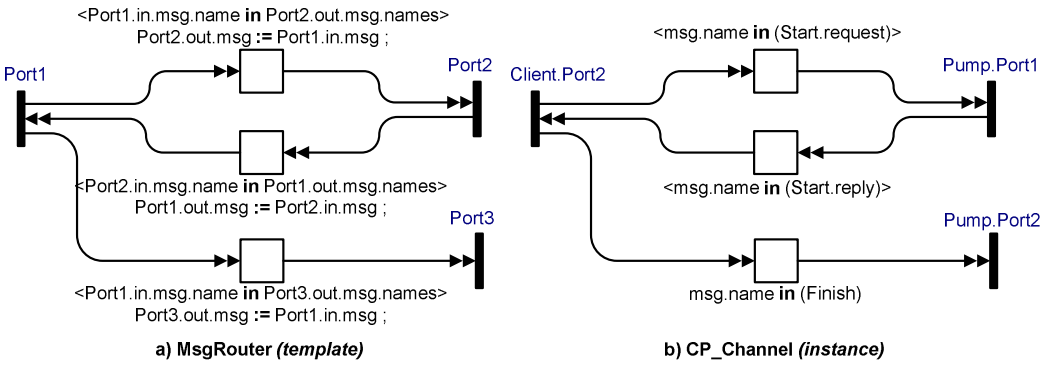

FIG. 6 – **L*f*P**-BD du modèle de média MsgRouter (a) et son instanciation dans le système (b).

Ce média est doté de 3 ports de communication : *Port1* et *Port2* connectables à des *binders* bidirectionnels (pour des requêtes RPC), et *Port3* unidirectionnel (pour des requêtes sans réponse). Le diagramme d'architecture logicielle (figure 3) spécifie comment ces ports seront connectés aux binders des classes frontières. Ce média assure le routage de l'information explicitement en fonction des noms des messages, et implicitement en fonction des destinataires spécifiés dans leur en-tête. Cette dernière information est implicite car, à la definition du média le nombre et le type des instances de classes connectées n'est pas connu.

La figure 6.b montre l'instanciation *CP_Channel* de ce média dans le système. Ses ports sont liés effectivement au *Port2* du **`client`** et aux *Port1 et Port2* de la **`pump`**. La connection aux classes frontières permet de définir quels messages peuvent être effectivement acceptés par les différents binders. En l'occurrence, ce média permet de véhiculer les requêtes *Start* (RPC) de la classe **`client`** sur le *Port1* de la classe **`pump`** et les messages *Finish* asynchrones (car spécifiés «oneway» sur le diagramme de classe UML de la figure 2) sur le *Port2* de la classe **`pump`**

Ce média exhibe deux caractéristiques intéressantes :

– La généricité de définition d'un média, qui permet de réutiliser des protocoles complexes déjà spécifiés. Le **`client`** connecté au *Port1* ne connaît pas explicitement l'existence des différents ports de la **`pump`**, et le média dans sa forme générique ne fait aucune hypothèse sur le contenu des messages qu'il va véhiculer.

– La possibilité pour une classe de disposer de plusieurs points de connexion de sémantiques différentes, ce qui permet d'élaborer facilement des synchronisations et traitements distribués entre classes **L*f*P**. Notre démarche se distingue du modèle événementiel d'UML où tous les messages sont délivrés à travers une unique file de taille non-spécifiée. Nous adoptons une description qui s'apparente à celle de tâches communicantes (à la manière d'Ada95), ou d'objets UML actifs.

5 Éléments pour la vérification avec L*f*P

Nous nous appuyons sur les réseaux de Petri (RdP) pour vérifier des modèles **L*f*P**. Les réseaux de Petri sont particulièrement adaptés pour exprimer le parallélisme et sont employés aussi bien en analyse d'algorithmes répartis, qu'en automatique ou en évaluation de performances. Nous utilisons ici les réseaux de Petri colorés, un modèle de haut niveau permettant de décrire de façon concise des systèmes comportant des éléments répliqués.

Un réseau de Petri est un graphe bipartite composé de places et de transitions. Une place, symbolisée par un cercle, est un élément de stockage qui peut contenir des jetons, elle est alors marquée. Une transition, symbolisée par un rectangle, est un élément actif représentant les évolu-

tions du système. Le franchissement d'une transition consomme de façon atomique des jetons des places en amont de la transition (places précondition), et en restitue dans les places en aval (places post-condition). Une transition est dite franchissable si ses places préconditions sont marquées. Le choix de la transition à franchir est indéterministe parmi les transitions franchissables du réseau.

On parle de réseaux colorés lorsqu'une "couleur", ou type, est associée aux jetons. Une place ne peut contenir que certaines couleurs, qui forment son domaine. Un domaine est créé à partir de la composition d'autres domaines de couleurs. Les arcs, reliant les places aux transitions possèdent une fonction (dite fonction de couleur), permettant de spécifier la nature des jetons pris ou mis dans une place lors du franchissement de la transition. Pour une présentation formelle de ce formalisme voir [8, 6].

Les informations comportementales de **L*f*P** peuvent être utilisées pour créer un réseau de Petri. Les différentes classes et médias sont traduites séparément, puis reliées pour former un modèle unique sur lequel les vérifications vont pouvoir être effectuées.

5.1 Exemple de traduction : la classe `pump`

La figure 7 présente la traduction en réseaux de Petri colorés de la classe **`pump`** présentée en figure 5. Chaque instance de pompe sera modélisée par un jeton qui porte la valeur des deux variables locales de la classe (**ID** du client, plafond **Credit**), ainsi qu'un identifiant unique permettant d'assurer le routage entre les instances de classes (nommage global des entités du système). Le domaine des jetons représentant une pompe est donc PumpTok composé de *<entities,client,money>*, correspondant aux variables d'instance *<monID,IDclient,Credit>*. Les jetons représentant les pompes sont initialement placés en S0, initialisés avec les valeurs par défaut (*<P1,error,0>*,*<P2,error,0>*).

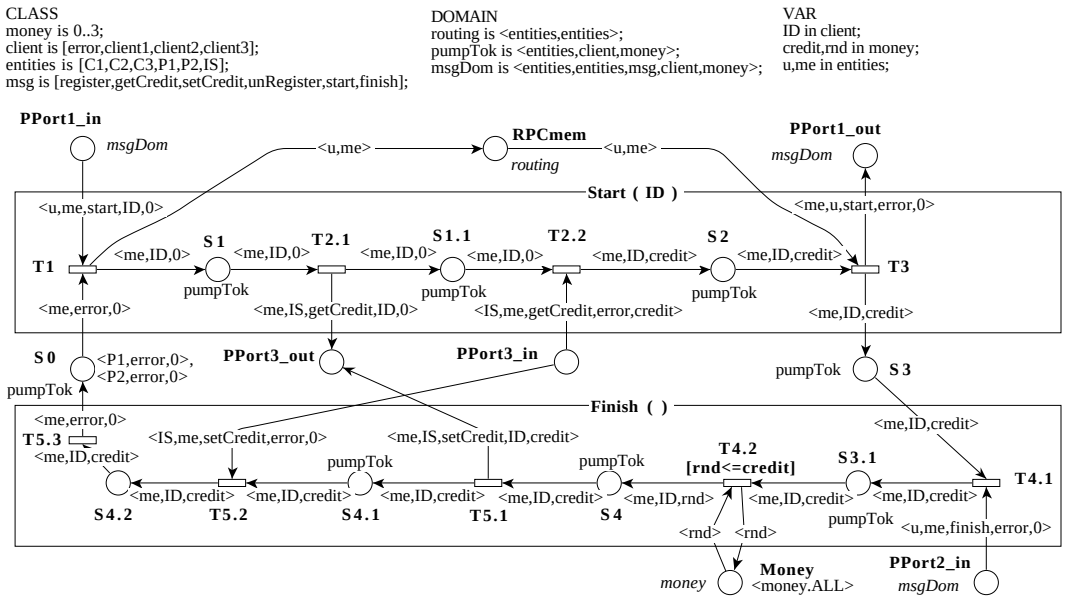

FIG. 7 – Modèle réseau de Petri de la classe **L*f*P `pump`**

La structure de l'automate de contrôle du **L*f*P**-BD est conservée : on retrouve les états `S0` à `S4` et des transitions `Ti.x` correspondant à la traduction de la transition **L*f*P** `Ti`, décomposée en pseudo-instructions élémentaires. Cette décomposition introduit des états intermédiaires notés `Si.x`.

Les binders sont ici représentés par des places d'où arrivent les messages des autres entités. Un mécanisme FIFO peut être implémenté mais n'est pas représenté ici pour éviter de surcharger

la figure. Les places représentant les binders correspondent à des places d'interface avec les autres entités du système, et permettront par fusion de connecter les parties du modèle. Comme les RdP sont fortement typés et que les medias doivent permettre de véhiculer différents types de messages, on peut soit multiplier le nombre de places générées (chacune gérant un type de message), soit uniformiser le type des informations à transmettre. C'est cette deuxième solution, plus compacte, que nous avons choisi ici. Le domaine des places binder est donc construit pour véhiculer tous les messages possibles, et est composé de <*source,destination,nom du message,argument1,argument2*>. Le mécanisme RPC permettant un *return()* anonyme sur l'arc liant T3 à *Pump.port1* dans la figure 5 est implémenté par la place **RPCmem**. Elle permet de stocker les identificateurs des requêtes en cours, et de retrouver l'émetteur pour lui adresser la réponse.

Une pompe **P** peut démarrer l'exécution de *Start* dès qu'une requête à son attention arrive par **PPort1_in**. On s'assure que le destinataire du message est bien celui qui le consomme en T1 (variable *me* identique sur les deux arcs). **P** lance alors la requête *getCredit* par **PPort3_out** et attend la réponse en S1.1. Sur réception de la réponse (T2.2), **P** est libre de franchir T3, ce qui renvoie un acquittement au client à l'origine de la requête par **PPort1_out**, et place **P** dans l'état intermédiaire S3.

L'exécution de la méthode *Finish* est déclenchée par la réception (T4.1) du message par **PPort2_in**. T4.2 correspond à l'instruction *Credit = RandomMax(Credit)*. Son franchissement prend aléatoirement dans la place Money une valeur *rnd*. La valeur choisie doit être inférieure a **credit**, comme l'indique la garde portée par T4.2 : **[rnd<=credit]**. La valeur *rnd* est affectée à **credit** (troisième élément du jeton), puis replacée dans la place Money afin de préserver son marquage (<money.ALL> soit un jeton par valeur du domaine). La transition T5 est décomposée en trois pseudo-instructions : envoi de la requête (T5.1), réception de la réponse (T5.2), et réinitialisation des variables **ID** et **Credit** ici fusionnées dans T5.3. Cette affection parallèle est autorisée car chaque instance de **pump** est la seule à avoir l'accès à ses variables d'instance.

5.2 Exemple de vérification : détection de comportement déviant

Nous nous sommes intéressés dans un premier temps à assurer la vivacité du système. Cette propriété très simple assure que le système ne comporte pas d'états bloquants (e.g. aucune transition n'est franchissable).

5.2.1 Vérification modulaire : événements observables.

Afin de réduire la taille du modèle tout en préservant la propriété de vivacité, nous avons utilisé plusieurs techniques de réduction dont l'agglomération inspirée de [3, 9] qui préserve les transitions observables du point de vue de l'ensemble du système. Les états intermédiaires augmentent fortement la taille du graphe des marquages accessibles (GMA) car ils introduisent des entrelacements. Pour déterminer si deux transitions T1 et T2 sont agglomérables, éliminant l'état intermédiaire S qui les sépare, il faut vérifier que :

- les variables modifiées par le franchissement de T2 ne sont pas accessibles en lecture par le franchissement d'une quelconque autre transition du réseau pendant que S est marquée ;
- réciproquement, les variables lues par T2 ne sont pas modifiables par le franchissement d'une quelconque autre transition du réseau pendant que S est marquée ;
- T2 est la seule transition franchissable par un jeton placé en S

Ces critères ont des conséquences simples qui peuvent s'interpréter comme suit :

- T1 et T2 peuvent être fusionnées si elles ne manipulent que des variables locales.
- De plus, si les files de réception ne sont pas partagées entre les entités (1 :ALL) elles peuvent être considérées comme des variables locales. Cependant comme les messages sortants

agissent potentiellement sur le reste du système, les files d'émission sont alors considérées comme des variables globales.

– Une action qui modifie l'environnement peut être agglomérée avec les actions locales qui la précédent en conservant l'ordre des opérations. Elle peut également être agglomérée avec les actions locales qui lui succèdent si celles-ci ne font pas d'appels à l'environnement (en particulier pas de lecture sur une file de réception).

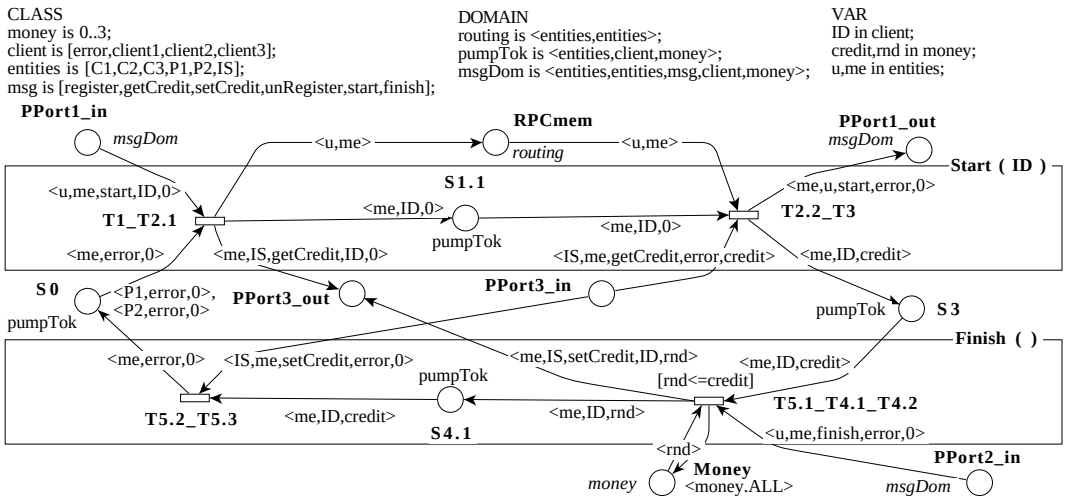

FIG. 8 – Modèle réseau de Petri de la classe **L*f*P `pump`** après agglomération

La figure 8 montre le modèle en RdP de la pompe après agglomération :

– Les transitions `T1` et `T2.1` sont agglomérables selon les règles énoncées ci-dessus, car il s'agit d'une reception dans une file non partagée (*PPort1_in*) suivie d'une emission. Cette opération diminue l'entrelacement et donc le nombre d'états, mais préserve l'indéterminisme global du système. En effet, la place `S1` permettait l'entrelacement suivant : **(a)** `T1(P1) -> T2.1(P1) -> T1(P2) -> T2.1(P2)` ou **(b)** `T1(P1) -> T1(P2) -> T2.1(P1) -> T2.1(P2)` ou **(c)** `T1(P1) -> T1(P2) -> T2.1(P2) -> T2.1(P1)`, ainsi que les scenarios symétriques en tirant `T1(P2)` en premier. Sur le modèle réduit on ne permet que le scenario **(a)** et son symétrique. Du point de vue du reste du système, la seule action observable est l'émission du message en franchissant `T2.1`. En conséquence, cette réduction préserve bien l'ensemble des scenarios observables *du point de vue du système*, et permet d'optimiser la vérification. Les transitions `T2.2` et `T3` sont agglomérables pour les mêmes raisons.

– L'agglomération est poursuivie jusqu'à stabilisation : `T4.1` et `T4.2` sont agglomérées (actions locales) en `T4.1_T4.2`, elle même agglomérée avec `T5` (actions locales précédant une emission). Bien que la table **Money** permettant le choix d'une valeur aléatoire soit une variable partagée entre les instances de pompes, son marquage reste fixe, ce qui fait de `T4.2` une action locale.

– Le marquage initial de `S0`, non nul, interdit de supprimer cette place.

Le gain de cette opération est combinatoire par rapport aux nombre d'instances de l'entité considérée.

5.2.2 Analyse des résultats

L'étude du modèle obtenu montre l'existence d'un interblocage qui arrête le système. Celui ci est dû à l'appel *finish* qui est asynchrone non-bloquant. Le marquage bloquant est obtenu quand le premier message de la FIFO *ISPort_in* est *setCredit(ID,sum)* et que l'**`infosystem`** ne peut le

consommer car le client identifié par *ID* a déjà quitté la station service. Ceci est dû au scenario dans lequel le client enchaîne les actions *Finish* et *getChange* suivi du *unregister* de l'opérateur (**`operator`**) avant que la pompe (**`pump`**) n'ait eu le temps d'effectuer un *setCredit*.

Ce scenario est possible, car en construisant le **L*f*P**-BD de l'**`infosystem`** à partir de la spécification UML, nous n'avons pas ajouté de contraintes sur l'ordre des appels à ses méthodes. Une correction consisterait donc à spécifier le caractère obligatoire de l'enchaînement des appels *register, getCredit, setCredit, unRegister* à l'aide d'un **L*f*P**-BD, et à permettre un traitement correct en donnant au binder **IS_Port** une sémantique de *multi-ensemble (bag)*, ne préservant pas l'ordre des messages.

Cette correction effectuée, le réseau devient vivant (7 millions d'états accessibles, 31 millions d'arcs). Les bornes structurelles nous permettent alors de dimensionner les tailles des files de messages. Ainsi *IS_Port* a une taille bornée par *Nombre d'instances de* ***`pump`*** *+ Nombre d'instances d'****`operator`***, *Operator_Port1* et *P_Port1* sont bornées par *Nombre de* ***`client`*** et *P_Port2* est bornée par une taille de 1.

6 Conclusion

Nous avons présenté une approche de prototypage par raffinements pour les systèmes répartis embarqués. Notre approche s'inspire de MDA (Model Driven Architecture), prônée par l'OMG [22]. Le système est décrit au moyen d'un modèle **L*f*P** que l'on peut construire à partir d'une spécification UML. **L*f*P** est un langage de spécification ayant les caractéristiques d'un ADL (Architecture Description Language) et permettant de capturer de manière non ambiguë les interactions entre composants d'un système réparti. Le modèle **L*f*P** sert de base a la vérification formelle du système à développer, et à la synthèse automatique du squelette de contrôle réparti de l'application.

Avec une telle approche, les squelettes de contrôle générés automatiquement sont conformes à leur spécification (propriété garantie par le générateur de code) et les propriétés vérifiées sur le modèle sont préservées. Cela permet non seulement de focaliser l'effort de conception sur un modèle dédié à la vérification et à la synthèse de programme, mais d'offrir également un haut niveau de fiabilité dans la phase de maintenance, cette dernière pouvant s'effectuer sur le modèle **L*f*P**.

L'application "à la main" des techniques présentées dans cet article a donné des résultats intéressants sur plusieurs études et permis de produire rapidement des spécifications formelles analysables. Une étape importante en cours est de construire les outils permettant d'automatiser l'usage de ces techniques pour en permettre une utilisation ne nécessitant pas la connaissances des méthodes formelles sous-jacentes.

Remerciements : Nous tenons à remercier Isabelle Mounier et Emmanuel Paviot-Adet pour leurs conseils dans le traitement de l'étude de cas présentée dans cet article.

Références

[1] J. Abrial. *The B-book.* Cambridge University Press, 1995.

[2] J. Araujo and A. Moreira. Specifying the behaviour of UML collaborations using Object-Z. In *Americas Conference on Information Systems (AMCIS)*. AIS, 2000.

[3] G. Berthelot. Checking properties of nets using transformations. *Advances in Petri Nets, vol. 222 LNCS*, 1985.

[4] R. Clark and A. Moreira. Use of E-LOTOS in adding formality to UML. *Journal of Universal Computer Science*, 6(11) :1071–1087, 2000.

[5] J. Corbett. Evaluating deadlock detection methods for concurrent software. *Software Engineering*, 22(3) :161–180, 1996.

[6] M. Elkoutbi and R. Keller. Modeling Interactive Systems with Hierarchical Colored Petri Nets. In *1998 Conference on High Performance Computing*, April 1998.

[7] A. Evans and A. Kent. Core Meta-Modelling Semantics of UML : The pUML Approach. In *Proceedings of UML'99*. IEEE Computer Society Press, 1999.

[8] C. Girault and R. Valk. *Petri Nets for Systems Engineering*. Springer Verlag, 2002.

[9] S. Haddad. A Reduction Theory for Coloured Nets. *LNCS : High Level Petri Nets. Theory and Application*, 1991.

[10] D. Helmbold and D. Luckham. Debugging Ada tasking programs. *IEEE Software*, 2(2) :47–57, March 1985.

[11] ISO. *Information Technology – Programming Languages – Ada*. ISO, February 1995. ISO/IEC/ANSI 8652 :1995.

[12] ITU-T. Open Distributed Processing, X.901, X.902, X.903 and X.904 standard. Technical report, ITU-T, 1997.

[13] J. Lilus and I. Porres Paltor. vUML : a tool for verifying UML models. In *14th IEEE International Conference on Automated Software Engineering*, 1998.

[14] Luqi, V. Berzins, M. Shing, R. Riehle, and J. Nogueira. Evolutionary computer aided prototyping system (caps). In *Technology of Object-Oriented Languages and Systems*, 2000.

[15] Luqi and J. Goguen. Formal methods : Promises and problems. *IEEE Software*, 14(1) :73–85, January / February 1997.

[16] M. Macona Kandé and A. Strohmeier. Towards a UML profile for software architecture descriptions. In *UML 2000 - Advancing the Standard.*, volume 1939 of *LNCS*, pages 513–527. Springer, 2000.

[17] S. Marc, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI : The Complete Reference*. MIT Press, 1996.

[18] N. Medvidovic and R. Taylor. A classification and comparison framework for software architecture description languages. *Software Engineering*, 26(1) :70–93, 2000.

[19] Sun Microsystems. *Java Remote Method Invocation Specification - SUN JDK 1.2*, October 1997.

[20] OMG. The common object request broker : Architecture and specification, revision 2.2. Technical report, OMG, 1998.

[21] OMG. OMG Unified Modeling Language Specification, version 1.3. Technical report, OMG, 1999.

[22] OMG. Model Driven Architecture (MDA), Document number ormsc/2001-07-01. Technical report, OMG, 2001.

[23] I. Paltor and J. Lilius. vUML : A tool for verifying UML models. In *Proc. of the 14th IEEE International Conference on Automated Software Engineering, ASE'99*. IEEE, 1999.

[24] D. Regep and F. Kordon. **L*f*P** : a specification language for rapid prototyping of concurrent systems. In *12th IEEE International Workshop on Rapid System Prototyping*, June 2001.

[25] J. Saldhana and S. Shatz. UML Diagrams to Object Petri Net Models : An Approach for Modeling and Analysis. In *Int. Conf. on Soft. Eng. and Knowledge Eng. (SEKE'2000)*, 2000.

[26] A. Singhai. *QuarterWare : A middleware toolkit of software RISC components*. PhD thesis, University of Illinois at Urbana-Champaign, 1999.

[27] J. Zhao. A slicing-based approach to extracting reusable software architectures. In *CSMR*, pages 215–223, 2000.