Un environnement de conception de systèmes distribués basé sur UML

Ludovic Apvrille*, Pierre de Saqui-Sannes**, ***, Renaud Pacalet*, Axelle Apvrille****

Résumé

Cet article propose un nouvel environnement de développement des systèmes distribués, basé sur le profil UML TURTLE. Aux étapes d'analyse et de conception qui firent l'objet de précédents articles, nous ajoutons une étape de déploiement. Il s'agit en l'occurrence de déployer des composants TURTLE sur des noeuds matériels d'exécution et de modéliser les liens entre ces noeuds d'exécution. A l'exemple des diagrammes TURTLE utilisés en analyse et conception, les diagrammes de déploiement se voient dotés d'une sémantique formelle par traduction vers le langage RT-LOTOS. L'outil TTool (TURTLE Toolkit) est enrichi d'un générateur de code exécutable Java capable de prendre en compte les composants TURTLE déployés sur des noeuds et les liens entre les noeuds d'exécution. TTool génère maintenant du code réseau qui utilise les protocoles de type UDP ou RMI pour assurer les communications entre composants. L'intrusion d'un pirate dans une session HTTP sécurisée sert d'exemple illustratif de l'environnement proposé.

Mots clés : systèmes distribués, UML, techniques de description formelles.

A UML-based Framework for Distributed System Design

Abstract

This paper introduces a new environment for developing distributed systems. It is based on the TURTLE UML profile. Analysis and design phases, described in previous papers, have been extended with an additional deployment phase. In this new step, TURTLE components are deployed over hardware execution nodes, and nodes are connected together throughout links. TURTLE deployment diagrams are given a formal semantics in RT-LOTOS, therefore following the approach used for TURTLE analysis and design diagrams. Moreover, the paper presents a Java code generator which outputs appropriate Java code for TURTLE deployment diagrams. This code is automatically deployable on networks because it implements node communication using network protocols such as UDP or RMI. TTool, The TURTLE toolkit has been extended to support these new diagrams and code generators. The attack of protected data exchanged throughout secured HTTP sessions serves as example.

Key words: distributed systems, UML, formal description techniques.

* Laboratoire System-On-Chip, GET / ENST, 2229 routes des crêtes, B.P. 193, 06904 Sophia-Antipolis Cedex, France, ludovic.apvrille@enst.fr, renaud.pacalet@enst.fr
** LAAS-CNRS, 7 avenue du Colonel Roche - 31077 Toulouse Cedex 4, France.
*** ENSICA, 1 place Emile Blouin, 31056 Toulouse Cedex 5, France, desaqui@ensica.fr.
**** MISC Magazine, axellec@miscmag.com

I. INTRODUCTION

Forte de ses treize diagrammes, la notation UML 2.0 [5] dispose en principe d'atouts suffisants pour couvrir les phases d'analyse, conception et déploiement de systèmes distribués. Dans les faits, UML 2.0 pêche par le pouvoir d'expression trop limité de ses diagrammes de déploiement. De plus, la notion de « point de variation sémantique » [5] confirme si besoin était l'absence de sémantique formelle « officielle » pour UML. Enfin, les ateliers UML 2.0 n'offrent aucune facilité particulière pour assister le/la concepteur/trice au long de la trajectoire analyse-conception-déploiement. Des outils de synthèse de conception mais aussi des outils de validation formelle de modèle et des générateurs de code exécutable prenant en compte l'aspect « distribué » des systèmes leur font défaut.

Ce constat nous a amené à développer le volet « distribution » d'un profil UML temps réel appelé TURTLE (*Timed UML and RT-LOTOS Environment*) [1]. Doté d'une sémantique formelle par traduction vers le langage RT-LOTOS [24], TURTLE est un profil UML temps réel outillé par la chaîne d'outils TTool [4]-RTL [25]. TTool comprend un éditeur de diagrammes et un générateur automatique de code RT-LOTOS. Il exploite l'outil RTL pour valider formellement les spécifications RT-LOTOS qu'il engendre à partir de modèles TURTLE.

Initialement centré sur le couple (architecture, comportements) qui sous-tend toute conception, TURTLE a été étendu pour couvrir d'une part, la phase d'analyse à base de scénarios et d'autre part, le regroupement des objets dans des composants eux-mêmes répartis sur des sites d'implantation.

En ce qui concerne la phase d'analyse, le profil TURTLE permet de décrire un système sous la forme de scénarios (diagrammes de séquences) structurés par un diagramme d'interactions [3]. Fait rare pour un outil UML, TTool permet de synthétiser automatiquement une conception TURTLE (diagramme d'objets et leurs comportements) à partir des diagrammes d'interactions et de séquences résultant de l'analyse.

En ce qui concerne la phase de déploiement distribué, un précédent article [2] avait amorcé le travail sur l'adjonction d'un volet « systèmes distribués » au profil TURTLE. Cette extension appelée TURTLE-P permet de regrouper les objets en composants eux-mêmes répartis sur des sites identifiés par les noeuds d'un diagramme de déploiement. Certes, les diagrammes TURTLE-P ont été dotés d'une sémantique formelle. Mais sur le plan de la génération de code pour applicatifs distribués, nous étions freinés par le caractère abstrait des liens entre nœuds des diagrammes de déploiement. Ainsi pouvions-nous exprimer un délai de transmission mais en aucun cas paramétrer un lien Ethernet.

La contribution de cet article se situe dans le renforcement du pouvoir d'expression des diagrammes de composants et de déploiement initialement définis en [2], dans la proposition de mécanismes de génération de code reposant sur les technologies TCP / UDP ou middleware, et dans l'intégration de ces technologies à l'outil TTool.

L'article est structuré de la manière suivante. Le paragraphe II introduit le profil UML TURTLE. Le paragraphe III présente la méthodologie que nous entendons appliquer aux systèmes distribués. Le paragraphe IV traite de la sémantique formelle du profil TURTLE étendu. La génération de code fait l'objet du paragraphe V. Au paragraphe VI, l'intrusion d'un pirate dans une session HTTP sécurisée sert d'exemple illustratif de l'approche proposée. Le paragraphe VII identifie certaines limitations de notre approche, et esquisse des solutions. Le paragraphe VIII positionne notre contribution par rapport aux travaux du domaine. Enfin, le paragraphe IX conclut l'article.

II. LE PROFIL UML TURTLE

En supposant les besoins utilisateurs déjà exprimés, nous démarrons la modélisation TURTLE par la construction d'un diagramme de cas d'utilisation (*Use Case Diagram*, UCD). Sans ajout par rapport à un UCD UML 2.0, un UCD TURTLE permet de délimiter le périmètre du système étudié et d'isoler ce système de ses acteurs externes. Typiquement, on va retrouver un cas d'utilisation par phase de protocole et des acteurs qui représentent respectivement les utilisateurs de la couche de protocole en question et le service sur lequel elle s'appuie.

Les cas d'utilisation vont guider la modélisation dans son ensemble. Il faut les documenter et pour cela nous utilisons un diagramme global d'interaction (*Interaction Overview Diagram* en anglais, ou IOD) et des diagrammes de séquences (*Sequence Diagram*, SD). Le premier permet de structurer les scénarios exprimés par les seconds. Le profil TURTLE ajoute aux IOD UML 2.0 un opérateur de préemption qui permet d'exprimer qu'un scénario peut à tout moment interrompre un autre scénario. Sans extension particulière par rapport aux SD UML 2.0, les SD TURTLE seront utilisés pour décrire le service (local ou global) entre les acteurs et le système. On éclate ensuite les

diagrammes obtenus pour inclure les entités de protocole et le service sous-jacent, franchissant ainsi le pas vers une conception basée, par exemple, sur une architecture à trois niveaux dans laquelle des entités de protocole s'appuient sur un service existant pour apporter leur valeur ajoutée sous forme de service à la couche supérieure.

Une conception TURTLE comprend un diagramme de classes/objets (*Class Diagram*, CD) pour définir la structure statique du système étudié et un ensemble de diagrammes d'activités (*Activity Diagram*, AD) utilisés pour décrire le comportement interne des objets. Les CD transposent à la composition d'objets UML l'idée d'opérateurs de composition chère aux algèbres de processus. Il est ainsi possible d'exprimer explicitement et sans ambiguïté sémantique le fait que deux objets s'exécutent en parallèle (avec ou sans synchronisation) ou bien s'exécutent en séquence. On peut également modéliser le fait qu'un objet peut en préempter un autre. En termes de comportement, les AD TURTLE ajoutent aux diagrammes d'activité d'une part des actions de synchronisation (les objets TURTLE communiquent en effet par rendez-vous) et des opérateurs temporels (délai fixe, délai non déterministe qui combiné au premier permet de travailler avec des intervalles temporels, offre limitée dans le temps pour éviter tout blocage infini sur une action synchronisée).

L'un des intérêts majeurs du profil TURTLE réside dans le fait qu'il est outillé. L'outil TTool (TURTLE Toolkit [4]) permet d'éditer les cinq types de diagrammes évoqués plus tôt (UCD, IOD, SD, CD, AD), (cf. Figure 2). TTool inclut également deux générateurs de code. Outre le générateur de code Java développé très récemment et qui figure parmi les contributions de cet article, TTool inclut un générateur de code RT-LOTOS. Un modèle TURTLE[1] est ainsi traduit dans une spécification RT-LOTOS qui peut à son tour être validée en utilisant l'outil RTL (Real-Time Lotos Laboratory, développé au LAAS-CNRS [25]). RTL offre des facilités de simulation et d'analyse d'accessibilité. Une interface avec CADP (développé à l'INRIA [26]) permet de minimiser les graphes d'accessibilité. Le point à retenir est que TTool rend l'usage de RT-LOTOS totalement transparent à l'utilisateur. Il gère en particulier les problèmes de remontée des résultats de simulation vers le modèle TURTLE.

La chaîne d'outils TTool-RTL-Aldébaran couvre ainsi les phases amont du cycle de vie en offrant la possibilité de confronter une conception aux besoins et propriétés identifiés en phase d'analyse. Une originalité de l'outil TTool est de proposer une synthèse automatique d'une conception TURTLE à partir d'une analyse TURTLE (IOD, SD). La conception est alors validée après traduction vers RT-LOTOS. Ajoutons à ceci qu'il est possible de valider un modèle exprimé uniquement par des scénarios (IOD, SS) sans passer par des diagrammes de conception (CD, AD).

Quelque soit le cas de figure (synthèse automatique de conception, conception manuelle, modélisation limitée à des scénarios), la validation formelle du modèle TURTLE n'est qu'un premier filtre d'erreurs qui ne dispense pas de réaliser des maquettes d'implantation. De plus, la validation dont nous venons de parler s'opère sur un système « centralisé » qui ne prend pas en compte le déploiement de composants logiciels sur des sites d'implantation. Ces points liés à la génération de code et au déploiement vont être améliorés par les extensions qui font le cœur de cet article.

III. METHODOLOGIE

1. Cycle

La méthodologie de notre profil TURTLE étendu avec une phase de déploiement comporte quatre étapes (cf. Figure 1) :

1. L'analyse comprend la réalisation d'un diagramme global d'interactions (IOD) et de diagrammes de séquences (SDs) qui peuvent être validés formellement. Cette phase d'analyse a pour objectif la réalisation d'un sous-ensemble des scénarios possibles au regard des fonctionnalités décrites dans le cahier des charges d'un système. Il ne s'agit en aucun cas de réaliser la description complète du système, même si la capacité d'expression de ces diagrammes d'analyse a été enrichie avec la version 2.0 d'UML.
2. La synthèse automatique d'une conception TURTLE produit une première conception sous la forme d'un diagramme de classes (CD) qui décrit d'une part, la structure du système sous la forme de tclasses (TURTLE classes), de tobjets (TURTLE objets), et de relations de composition entre des éléments, et d'autre part le comportement du système sous la forme de diagramme d'activités qui modélisent les comportements des tclasses et tobjets du CD. Le concepteur peut alors faire évoluer cette conception et vérifier formellement le bon comportement du système. L'évolution de cette conception peut se faire de façon « guidée » en s'assurant, par utilisation de techniques de bisimulation, que la conception à l'étape n+1 possède les mêmes

[1] Un modèle TURTLE est soit une « analyse TURTLE », c'est à dire une IOD et plusieurs SD, soit une « conception TURTLE », c'est à dire un CD et plusieurs ADs.

ensembles de trace qu'à l'étape n, lorsque bien entendu les traces de l'étape n+1 sont épurées des actions non présentes à l'étape n. Cette bisimulation s'effectue ainsi entre le graphe d'accessibilité de l'étape n, et le graphe d'accessibilité de l'étape n+1 minimisé aux actions de l'étape n.

3. Les classes / objets TURTLE obtenus à l'étape 2 doivent être manuellement regroupés en composants, que nous appelons *tcomposants* (ou composant TURTLE). Parallèlement, un diagramme de déploiement (DD) définissant des noeuds matériels d'exécution doit être élaboré. Chaque tcomposant peut alors être projeté sur un ou plusieurs noeuds. La projection sur un noeud d'un tcomposant signifie qu'une instance de ce tcomposant sera exécutée sur le noeud correspondant. Si un tel tcomposant est projeté sur plusieurs noeuds, cela signifie que plusieurs instances de ce tcomposant fonctionneront dans le système. Enfin, le concepteur doit identifier les liens de communication entre les noeuds d'exécution, et préciser leur sémantique par caractérisation de certains paramètres, tels que le délai ou le taux de perte.
4. L'étape suivante consiste à générer automatiquement du code Java « réseau », c'est à dire générer un ensemble de paquetages Java qui sont susceptibles de communiquer entre eux par des protocoles de communication tels que UDP, TCP ou RMI. A la suite de la génération automatique de code Java, le développeur peut exécuter les différents paquetages logiciels générés. Ce code généré a comme objectif de fournir un prototype exécutable, qui a généralement l'avantage de respecter les propriétés validées formellement. Ce point est discuté plus en détail à la section VII.

Notons que cette méthodologie repose sur une approche incrémentale. A chaque étape, la sémantique formelle du profil permet de réaliser des validations formelles, et éventuellement de faire des preuves de conservation de propriété par utilisation des techniques de bisimulation. De plus, si les passages de l'analyse à la conception, et du déploiement à l'exécution sont automatisés, notre méthodologie n'offre par contre aucun support automatique pour l'aide à la construction des composants, des nœuds d'exécution ou pour le déploiement de ces composants. Nous discutons de ce point plus en détail dans la section VII.

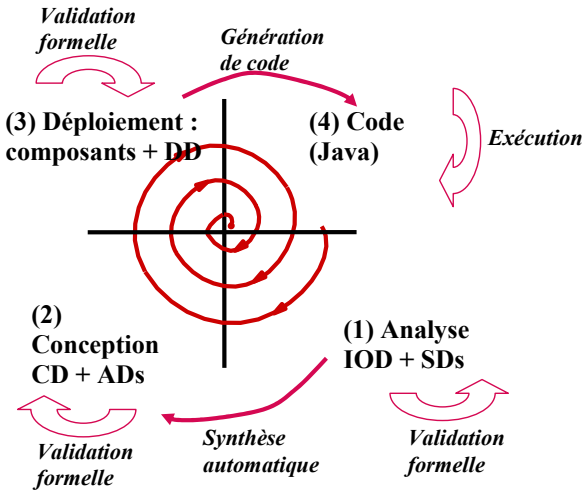

Figure 1. Méthodologie TURTLE pour les systèmes distribués / *TURTLE Methodology for distributed systems*

Deux règles simplifient l'identification des composants TURTLE à projeter sur les noeuds d'exécution :

1. Un composant TURTLE est indissociable. Un composant TURTLE est constitué de tclasses, de tobjets et de relations de composition. Le dire indissociable, c'est dire qu'à l'exécution, il n'est pas possible d'instancier un sous-ensemble de ce composant. Il forme un tout, indivisible, offrant toujours la même interface à son environnement, ce qui ne serait pas le cas s'il était dissociable.
2. Une projection d'un composant TURTLE ne peut pas migrer d'un site d'exécution à un autre : elle ne peut être exécutée que sur le site d'exécution auquel elle a été initialement affectée.

2. Déploiement de composants TURTLE

Les éléments graphiques de nos diagrammes sont :

- des **noeuds d'exécution**, nommés par leur nom Internet ;

- des **composants TURTLE (*tcomposants*)**, représentés sous la forme d'artéfacts UML 2.0, nommés par leur nom de conception TURTLE, et dotés de référence à leur bibliothèque logicielle (voir section *Outillage*).
- des **liens de communication** qui sont munis à la fois de paramètres utilisés pour la génération de code formel (association entre portes de communication, délai, taux de perte) et de paramètres liés à la génération automatique de code exécutable, tels que le protocole utilisé (TCP, UDP, RMI). Ces liens sont dotés d'une sémantique asynchrone (la section IV sur la sémantique formelle explique plus en détail cet aspect).

3. *Outillage*

Comme expliqué dans la section de présentation du profil TURTLE, TTool permettait déjà l'édition des diagrammes d'analyse et de conception, et la génération automatique de code formel (RT-LOTOS) ou non formel (Java) et enfin l'exploitation des résultats de validation (cf. Figure 2). L'outil permet à présent l'édition des diagrammes de déploiement. Il permet non seulement de modéliser des composants TURTLE mais aussi de les déployer. TTool permet de générer du code formel (RT-LOTOS) ou bien du code Java depuis les diagrammes de déploiement. TTool permet bien entendu de piloter ces outils externes et de manipuler de façon graphique les résultats du déploiement.

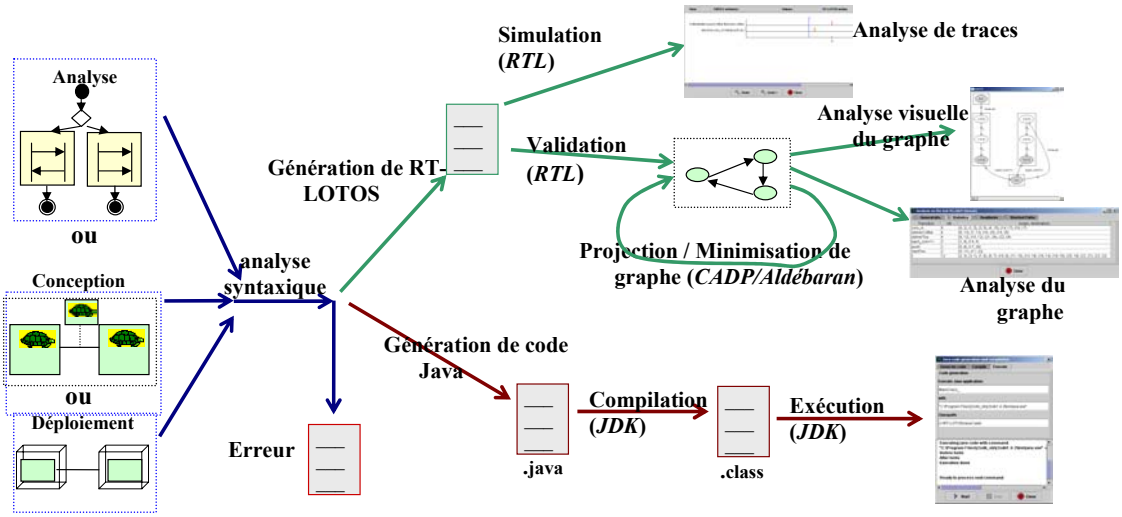

Figure 2. Nouvelles fonctionnalités de TTool pour la génération de code formel et informel / New *functionalities offered by TTool for the generation of formal and informal code*

Sous TTool, les composants TURTLE doivent chacun être modélisés dans une « conception TURTLE ». Par exemple, supposons que l'on désire modéliser un système distribué comprenant notamment une entité *client* et une entité *server*. Dans une première conception, que l'on pourrait intituler « Global System » (cf. Figure 3), nous allons modéliser un diagramme de classes (CD) comprenant des classes pour *client* et d'autres pour *server*, ainsi que les diagrammes d'activités correspondants.

Pour déployer ce système distribué sur des nœuds d'exécution, nous créons un nouveau diagramme de classes intitulé « PackageClient » et comportant le sous-ensemble des classes de CD relatives à *client*. De même, pour *server*, nous créons « PackageServer », un diagramme de classes constitué du sous-ensemble des classes de CD relatives à *server*. Ces deux diagrammes de classes représentent chacun un composant à déployer. Nous créons ensuite un diagramme de déploiement auquel nous ajoutons des nœuds d'exécution. A chacun de ces noeuds, il est possible d'affecter des « artéfacts » qui représentent chacun une instance d'exécution d'un composant TURTLE. Par exemple, si l'on crée un noeud d'exécution « PCClient », il est possible d'ajouter à ce noeud un artéfact PackageClient. Cela signifie que l'on souhaite exécuter un composant de type « PackageClient » sur ce noeud.

La Figure 3 illustre ce type de modélisation avec TTool. On constate qu'il existe un onglet de conception pour le

« Global System », un onglet de conception pour chaque composant et, enfin, un onglet de déploiement comprenant le diagramme de déploiement. Ce dernier comporte deux noeuds stéréotypés « PC » (le stéréotype est donné à titre de commentaire). Chaque noeud a en outre été nommé avec le nom Internet de la machine sur laquelle nous souhaitons exécuter le composant (par exemple, le noeud *voyager45*). Enfin, sur le noeud *voyager42*, nous avons ajouté un composant TURTLE référençant le composant *PackageClient* et, sur le noeud *voyager45*, un composant TURTLE référençant le composant de type *PackageServer*. Pour chacun des nœuds et pour chaque composant, nous avons spécifié le nom du package Java à générer. Ainsi, pour l'artéfact *PackageClient* du nœud *voyager42* le code Java généré par TTool sera stocké dans une archive java nommée *client.jar*.

Enfin, entre les deux noeuds d'exécution, nous avons introduit un lien de communication. Ce lien a été muni d'informations de génération automatique de code : le protocole de communication utilisé (TCP) ainsi que des paramètres concernant ce protocole (des numéros de port). Les autres choix possibles pour les protocoles étaient UDP et RMI. Le lien précise aussi deux associations de porte. Les protocoles proposés par TTool reposent sur des approches client/serveur, c'est à dire que l'une des entités (le client) initie la communication vers une entité (le serveur) qui est en attente de connexion. La flèche des liens permet de désigner l'entité serveur du lien. Bien entendu, comme dans les architectures client/serveur, des données peuvent être échangées du client vers le serveur, et du serveur vers le client.

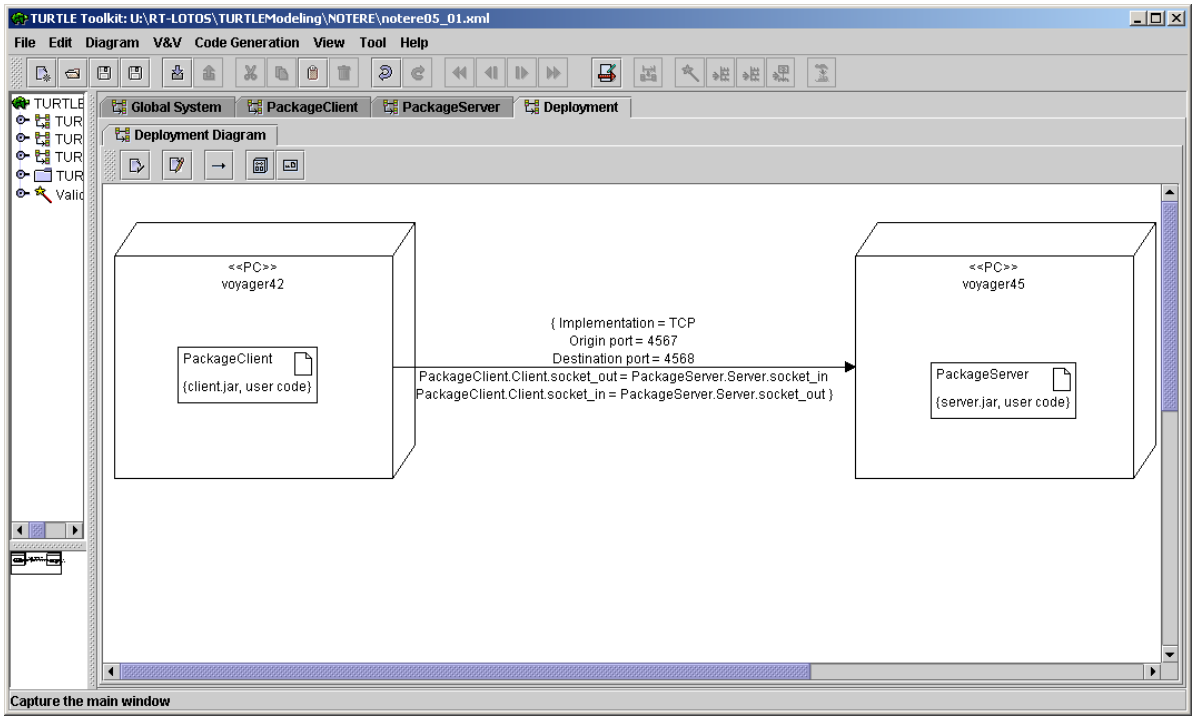

Figure 3. Diagramme de déploiement édité sous TTool / *Deployment diagram edited with TTool*

IV. SEMANTIQUE FORMELLE

La sémantique des diagrammes de déploiement est basée sur le profil TURTLE natif constitué des diagrammes de classes et d'activités, profil qui dispose lui-même d'une sémantique formelle exprimée par traduction vers RT-LOTOS.

Les éléments du diagramme de déploiement qui sont pertinents sont les composants et les liens[2]. En ce qui concerne la sémantique, il est important de connaître la nature de la communication entre les différentes classes des différents composants. Si les tclasses / tobjets appartiennent au même composant TURTLE alors la communication

[2] Les nœuds sont utiles pour le déploiement de l'application proprement dit, c'est à dire pour le déploiement des bibliothèques logicielles. Mais ils n'ont aucun rôle en ce qui concerne la sémantique. C'est pourquoi nous les ignorons ici.

est précisée par un opérateur de composition interne à ce composant. Si elles n'appartiennent pas au même composant alors la communication est asynchrone, et ses caractéristiques sont fournies au moyen des paramètres dont sont munis les liens entre les noeuds. Les liens sont considérés en outre comme totalement ordonnés du point de vue temporel. Cela a deux conséquences. La première est que deux messages émis au même instant ne sont pas ordonnés. La deuxième est que deux messages émis à des instants différents sont ordonnés selon leur instant d'émission, et bien entendu, si le délai associé au lien est déterministe. Par exemple, le délai du lien est [t0..t1], avec t1 >= t0. On suppose qu'un message m2 est émis au temps t2, et un message m3 au temps t3, avec t2 < t3. Si t1-t0<t3-t2, alors m2 est acheminé à destination avant m3. Sinon, il est possible que m3 arrive à destination avant m2.

La traduction des diagrammes de déploiement en RT-LOTOS, ce qui inclut donc la traduction des liens et les composants, se fait donc d'abord en générant des diagrammes TURTLE dits natifs (i.e. un diagramme de classes et des diagrammes d'activités TURTLE) puis en utilisant le générateur de code RT-LOTOS développé pour les conceptions TURTLE. Notre approche pour générer des diagrammes TURTLE fondamentaux depuis les diagrammes de déploiement est la suivante :

- Pour chaque composant TURTLE du diagramme de déploiement, on génère un nouveau nom (*artifact__xyz*).
- Les classes de chaque composant *artifact__xyz* sont, elles aussi, renommées (*nom_classe__artifact__xyz*). Les classes ainsi renommées sont copiées dans le diagramme de classes fondamental. Le diagramme d'activités de chaque classe est recopié dans l'ensemble des diagrammes fondamentaux.
- Pour chaque lien du diagramme de déploiement une classe le modélisant est également générée dans le diagramme de classes. Cette classe est construite comme suit. Pour chaque porte listée dans les paramètres du lien, elle possède une porte nommée *artifact__xyz__gateName*. Le diagramme d'activités de la classe relative à cette porte est analysé afin d'extraire tous les échanges de données possibles sur cette porte. Notons que ces échanges ne peuvent avoir lieu que dans un sens à la fois ; nous ne tolérons pas, une émission et une réception de donnée simultanées sur la même action, ce qui serait contraire à notre sémantique asynchrone des liens de communication. Bref, pour chaque type d'échange, nous générons une action correspondante dans le diagramme d'activités de la classe qui modélise le lien.

TTool permet de générer du RT-LOTOS directement depuis le diagramme de déploiement, sans laisser transparaître le passage par la génération automatique de ces diagrammes de classes et d'activités intermédiaires. La correspondance des actions décrites sur le graphe d'accessibilité aux actions des composants du diagramme de déploiement est listée par TTool.

V. GENERATION DE CODE

1. La génération de code depuis une conception TURTLE

La génération de code Java depuis une conception TURTLE nécessite de pouvoir traduire la structuration décrite sous la forme de classes TURTLE en une structuration sous la forme de tâches Java, et de pouvoir traduire le comportement des classes TURTLE sous la forme d'un comportement de tâches Java. Les tâches, au sens Java, sont vues comme des *Threads* Java. Les relations entre classes de type « séquence », « préemption », « parallélisme » se traduisent par des relations entre ces threads Java. De même, les comportements de ces classes, tels que les structures de contrôle de type choix déterministe, boucles, etc. se traduisent assez aisément en Java. Il reste donc à traiter le cas des opérateurs temporels, des actions sur des portes de synchronisation et des choix non déterministes.

Au niveau des diagrammes d'activités, des opérateurs temporels permettent de décrire des temps d'attente en spécifiant un délai déterministe ou un délai non déterministe. Ces mêmes opérateurs sont fréquemment utilisés pour modéliser de manière abstraite la durée d'exécution d'un algorithme dont on ne souhaite pas expliciter le code. Quand on en vient à la génération de code Java, il faut bien évidemment remplacer le délai de traitement d'un algorithme par la portion de code qui implante l'algorithme. Dans le cas où le délai est un vrai délai, et non la modélisation du temps de traitement d'un algorithme, l'opérateur doit être traduit tel quel par le générateur de code Java : pour ce faire, nous introduisons une attente équivalente dans le code Java. Mais dans le cas où le délai modélise un algorithme, le code Java correspondant à ce délai n'a pas de sens : il s'agit d'exécuter un algorithme. Pour cela, TTool autorise l'insertion de code Java au sein d'un opérateur temporel dont la signification temporelle sera ignorée en phase de génération de code Java : nous n'utilisons alors pour cet opérateur que le code fourni par

l'utilisateur. Bien entendu, en phase de validation formelle, le code Java utilisateur est ignoré au profit dudit opérateur temporel.

Quant aux actions de synchronisation, leur traduction est loin d'être évidente (sémantique TURTLE). Nous avons programmé des bibliothèques Java de synchronisation sur lesquelles le générateur Java de TTool s'appuie. Ces bibliothèques autorisent les actions au sens TURTLE, c'est à dire des actions synchronisées ou non, et des actions avec échanges bidirectionnels de données. Si certains opérateurs du langage Java ont simplifié l'implémentation de ces actions (l'opérateur *synchronized* par exemple), le passage des références de porte d'un thread à l'autre (par exemple, lors d'une séquence interne à une classe TURTLE, les portes de synchronisation sont passées d'une tâche à l'autre) a été complexe à implémenter, et a nécessité la mise en place de structures de données lourdes à instancier et à gérer. Le code ainsi généré n'a pas pour vocation l'étude de performances, mais plutôt la simulation. Les aspects « performance » seront traités dans de futures contributions, et sans doute avec d'autres langages, le C par exemple, tout en s'appuyant sur des interfaces de type POSIX.4.

Notons aussi qu'il est possible de spécifier du code Java à certains opérateurs, et notamment aux opérateurs d'action sur des portes. En ce qui concerne ces derniers, il est possible d'ajouter du code appelé « *precode* » et du code appelé « *postcode* ». Le code *precode* est exécuté avant l'action sur la porte, alors que le code *postcode* est exécuté une fois l'action sur la porte effectuée (cf. Figure 4). Ces codes Java insérés sont bien entendu totalement ignorés par le générateur de code RT-LOTOS. Dans ces conditions, et en ignorant les éventuelles faiblesses des générateurs de code Java et RT-LOTOS, il est bien entendu possible que la sémantique du code généré soit différente de celle du système validé.

La Figure 4 présente une modélisation TURTLE dont une action sur la porte « hello » a été enrichie par du code Java. Par la suite, du code Java a été généré pour cette modélisation TURTLE. Il a été spécifié qu'une unité de temps TURTLE est représentée, dans le code Java généré, par 10 millisecondes. Ce code a été par la suite compilé, puis exécuté, depuis TTool. Le résultat de cette exécution est présenté dans la fenêtre la plus à droite de la Figure 4.

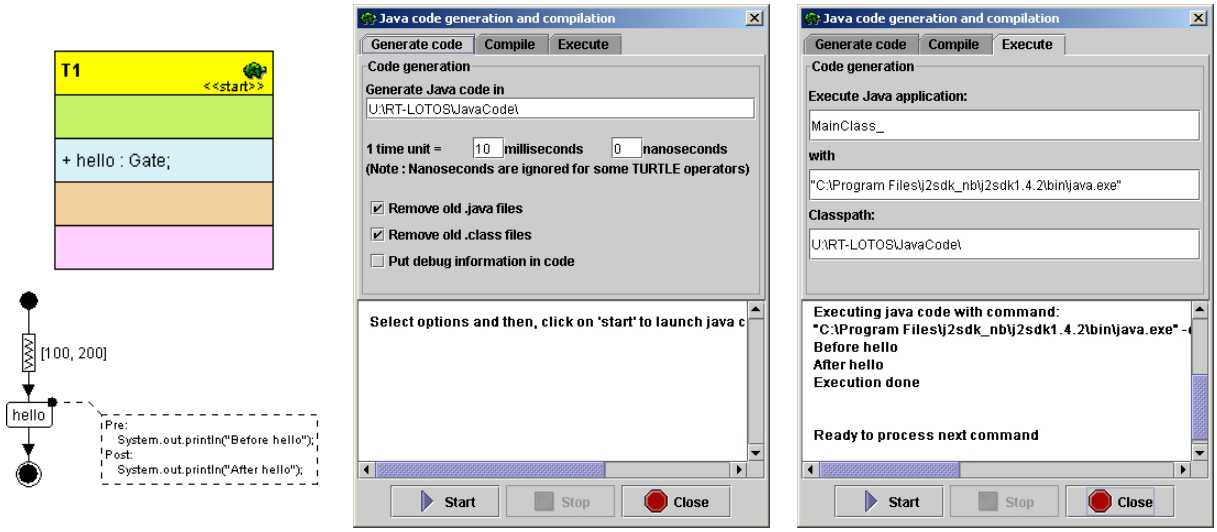

Figure 4. Modélisation TURTLE comportant du code Java ; Génération de code Java ; Exécution du code généré / *TURTLE modeling with Java code ; Generation of Java code ; Execution of generated code*

2. La génération de code depuis un déploiement TURTLE

TTool traduit les diagrammes de déploiement sous forme d'une conception TURTLE avant de générer du code RT-LOTOS. Pour le code non formel, nous avons réutilisé le même principe. Ainsi, le diagramme de déploiement est d'abord traduit sous la forme d'une conception TURTLE avant de générer du code Java associé à cette conception. Toutefois, cette approche présente trois principaux problèmes :

1. La conception possède les mêmes informations formelles que celles présentes dans le diagramme de déploiement (par exemple, le temps d'acheminement sur un lien), mais en aucun cas, les informations non formelles (par exemple le protocole utilisé).

2. Les classes qui modélisent les liens de communications ont été introduites afin de pouvoir valider formellement le comportement du lien. Mais aucun code ne doit être généré pour ce type de classes vu qu'aucun module logiciel à générer ne lui correspond.
3. Les communications entre les composants TURTLE se traduisent par des actions de synchronisation au niveau d'une conception TURTLE. Pourtant, en ce qui concerne le générateur de code, cela doit se traduire par des communications avec un protocole de type UDP, TCP ou RMI.

Afin de prendre en compte ces problèmes, nous avons dû adapter le générateur de code. Les trois problèmes cités précédemment ont été résolus en étendant la notion d'action de synchronisation dans le générateur de code et en modifiant le générateur de conception TURTLE depuis un diagramme de déploiement.

Nous avons ainsi introduit la notion d'action de synchronisation annotée. Si une action est annotée, alors elle est forcément liée à un protocole de communication (UDP, TCP, RMI), et son échange de données ne peut être que monodirectionnel. Cette annotation nous a obligé, d'une part à introduire un nouveau générateur de conception TURTLE et, d'autre part, à modifier la notion de conception TURTLE fondamentale et le générateur de code de cette conception TURTLE fondamentale (Figure 5).

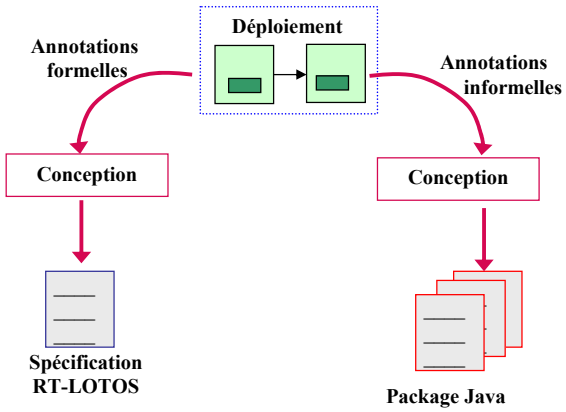

Figure 5. Processus de génération de code formel / informel depuis un déploiement TURTLE / *Formal and informal code generation process from a TURTLE deployment*

Dans le cadre de la génération d'une conception annotée avec des informations informelles, la conception fondamentale TURTLE ne comporte pas les classes modélisant les liens. A la place, les actions de synchronisation sont annotées comme « protocole ». De point de vue Java, le code correspondant à une action annotée est un appel à une bibliothèque réseau TURTLE. Cette bibliothèque permet l'ouverture de socket UDP / TCP sur des ports ou alors l'appel à une méthode distante sur l'objet cible via le middleware RMI.

Plus précisément, prenons le cas de la Figure 3. L'appel à l'action socket_out dans la classe Client du package client.jar se traduit par un appel soit à la fonction *sendTCP*() de notre bibliothèque dans le cas d'un envoi de données, soit un appel à *receiveTCP*() dans le cas d'une réception de données. Bien entendu, la bibliothèque gère les problèmes d'ouverture de connexion si besoin est, de retour des informations vers la même adresse IP que l'adresse source, etc.

Notons enfin qu'en ce qui concerne les diagrammes de déploiement, les nœuds sont annotés avec un nom de machine. Toutes les machines sont référencées avec ces noms là dans le code Java généré. Par exemple, dans le cas d'un envoi de données de la part de client.jar sur un *sendTCP*(), cet envoi se fera toujours vers la machine *voyager45*. C'est une limite de notre approche qui ne permet pas la migration de code.

Le générateur de code Java prend aussi en compte le code utilisateur ajouté sur certains éléments des diagrammes d'activités des composants TURTLE (*precode*, *postcode*). Pour chaque composant modélisé sur un diagramme d'activités, deux options sont disponibles sous TTool : ignorer ce code Java utilisateur, ou le prendre en compte. Dans le premier cas, l'annotation *user code* doit être spécifiée au niveau du composant. Dans le cas contraire, c'est l'annotation *no user code* qui est utilisée.

Comme dit précédemment, des bibliothèques réseaux programmées en Java sont fournies avec TTool. Les parties de ces bibliothèques réseaux spécifiques à un protocole donné ne représentent que quelques petites centaines de lignes de code. Ajouter un nouveau protocole à ces bibliothèques est une tâche plutôt aisée. Il s'agit principalement

de déclarer le protocole dans la bibliothèque en lui attribuant un numéro d'identification et de programmer des fonctions du type *sendMyProtocol*() et *receiveMyProtocol*(). Il convient aussi de modifier l'interface graphique de TTool pour ajouter, dans la liste des options des liens des diagrammes de déploiement, la sélection du protocole et de ses éventuels paramètres.

VI. ETUDE DE CAS

Le système distribué que nous proposons de traiter est celui d'un système client / serveur web, et d'un tiers malicieux. Le client communique avec le serveur suivant le mécanisme d'authentification basique du protocole HTTP [21]. Le tiers écoute les requêtes émises par le client vers le serveur et, par un mécanisme de rejeu, tente d'obtenir des données protégées. Cet exemple est purement didactique car le mécanisme d'authentification basique de HTTP est connu depuis des années comme étant sensible au rejeu.

1. Analyse du système

Le diagramme de cas d'utilisation de la Figure 6 est celui d'un système qui transfère des données entre un serveur et un client, avec ou sans mécanisme d'authentification. Nous supposons ici qu'un attaquant (*Attacker*) espionne le réseau pour récupérer les informations nécessaires au rejeu (cas d'utilisation « Retrieve authentication data »). Après avoir récupéré ces informations, l'attaquant peut émettre une requête vers le serveur lui permettant de récupérer les informations protégées (cas d'utilisation « Data transfer with authentication » et « Data Transfer »).

Par la suite, nous appelons client (*Client*) non pas l'utilisateur humain du navigateur, mais l'entité logicielle protocolaire du client. Idem pour l'attaquant, le serveur (*Server*) et l'entité réseau (*AttackedNetwork*).

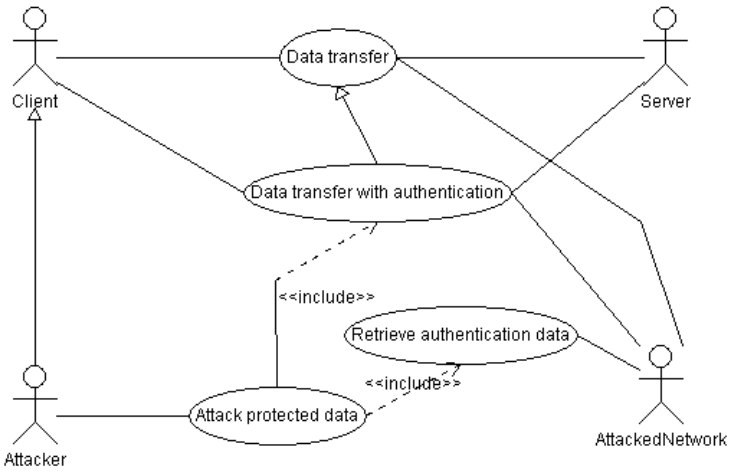

Figure 6. Diagramme de cas d'utilisation / *Use case diagram*

Le diagramme global d'interactions de la Figure 7 met en évidence les scénarios (non décrits ici) du protocole d'échange de données par HTTP avec et sans authentification basique [21]. Le client demande une page de données au serveur. Supposons cette page protégée (mécanisme d'authentification basique de HTTP) ; le client doit choisir entre deux options (premier choix non déterministe du diagramme d'interactions). La première consiste à demander cette page auprès du serveur en joignant ses données d'authentification à la requête *GET du protocole* HTTP et, dans ce cas, le serveur lui renvoie la page ou lui signale que ses données d'authentification sont erronées (deuxième choix non déterministe) lors du scénario *AuthAnsFromServer*. La deuxième option est de ne pas joindre de données d'authentification à la requête *GET* ; le serveur lui renvoie alors une erreur lui signifiant que cette page nécessite une authentification basique. A cette étape d'analyse, nous n'avons pas souhaité faire figurer le tiers malicieux ; il s'agit plutôt d'analyser le protocole HTTP afin de comprendre ses principaux mécanismes d'échange lors d'accès à des données protégées.

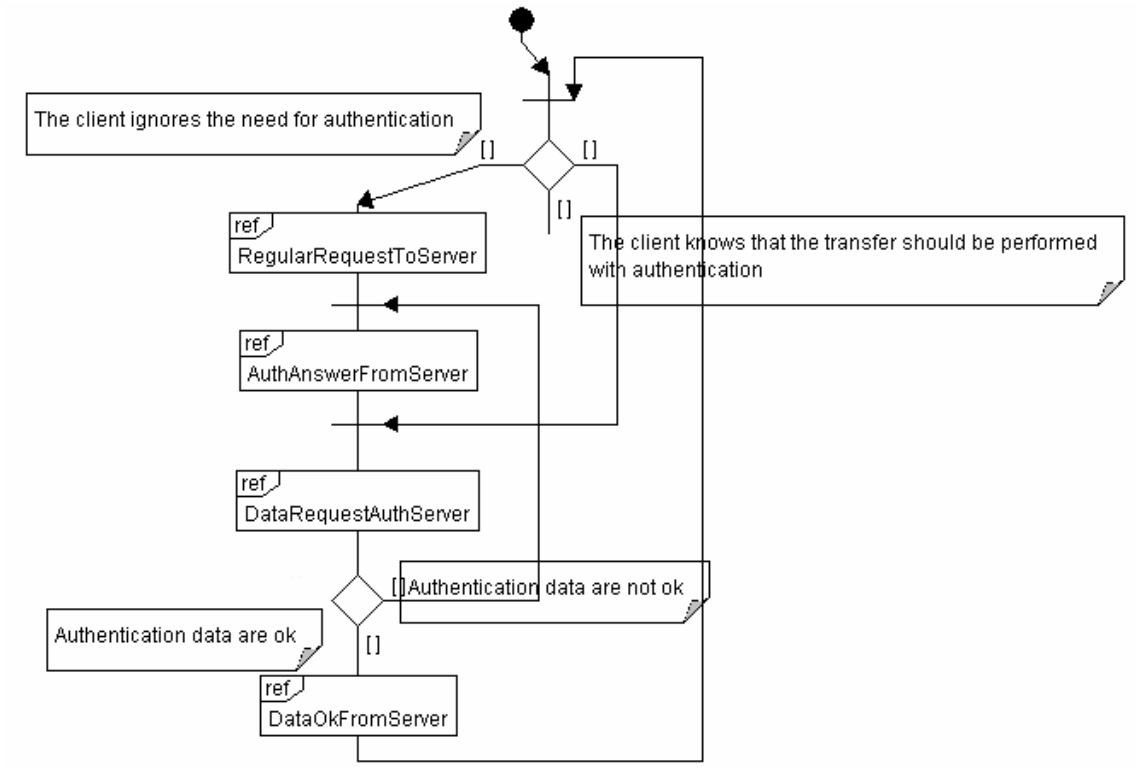

Figure 7. Diagramme global d'interactions pour un système client / serveur communicant par le protocole HTTP, avec et sans authentification / ***Interaction overview diagram for a client / server system communicating using HTTP, with and without authentication***

La génération d'une spécification RT-LOTOS depuis les diagrammes globaux d'interactions et de séquences a montré l'absence de situation d'interblocage et l'impossibilité pour le client de recevoir une donnée protégée sans avoir envoyé un *GET* accompagné des données d'authentification.

2. Conception du système

Nous avons généré automatiquement une première conception à partir des diagrammes d'analyse précédents. Cette première conception comporte une classe modélisant le client et une classe modélisant le serveur. Nous avons enrichi cette conception par un service réseau, dont le port d'émission du client peut être écouté par un attaquant, et par une classe modélisant cet attaquant (*Attacker*, cf. Figure 8). Selon le principe du rejeu, il émet les trames vers le serveur en déclarant sa propre adresse comme adresse de retour. S'il parvient à obtenir une donnée qui aurait nécessité une authentification, il réalise une action *attackSuccessful*. Sinon, il réalise une action *attackFailed*.

Le graphe d'accessibilité de ce système possède 57 états et 76 transitions. Sa minimisation sur les actions de l'attaquant conduit à un automate quotient de 6 états et 6 transitions (cf. Figure 9). Considérons le chemin de cet automate dans lequel l'attaquant intercepte une trame à destination du serveur. L'action port2_out est accompagnée d'un échange de données exprimé sur le graphe sous la forme <0, 1, 0, 54, 83>. La première donnée (0) modélise le fait que le paquet va à destination du serveur alors que la deuxième donnée (1) modélise le fait que le paquet provient du client. L'action 0 signifie que le client requiert la page située à l'adresse « 83 » et envoie comme données d'authentification « 54 ». Ces entiers sont bien entendu des modélisations des données réelles. L'attaquant veut lui aussi obtenir la donnée située à l'adresse « 83 ». Pour cela, il effectue un rejeu, en envoyant vers le serveur la même trame : il se contente de modifier l'adresse de retour de 1 à 2. L'attaquant reçoit alors en retour une trame envoyée par le serveur et qui contient la donnée protégée située à l'adresse « 83 » (cette donnée protégée a comme valeur « 67 »). L'attaque a bien eu lieu (action *attackSuccessful*).

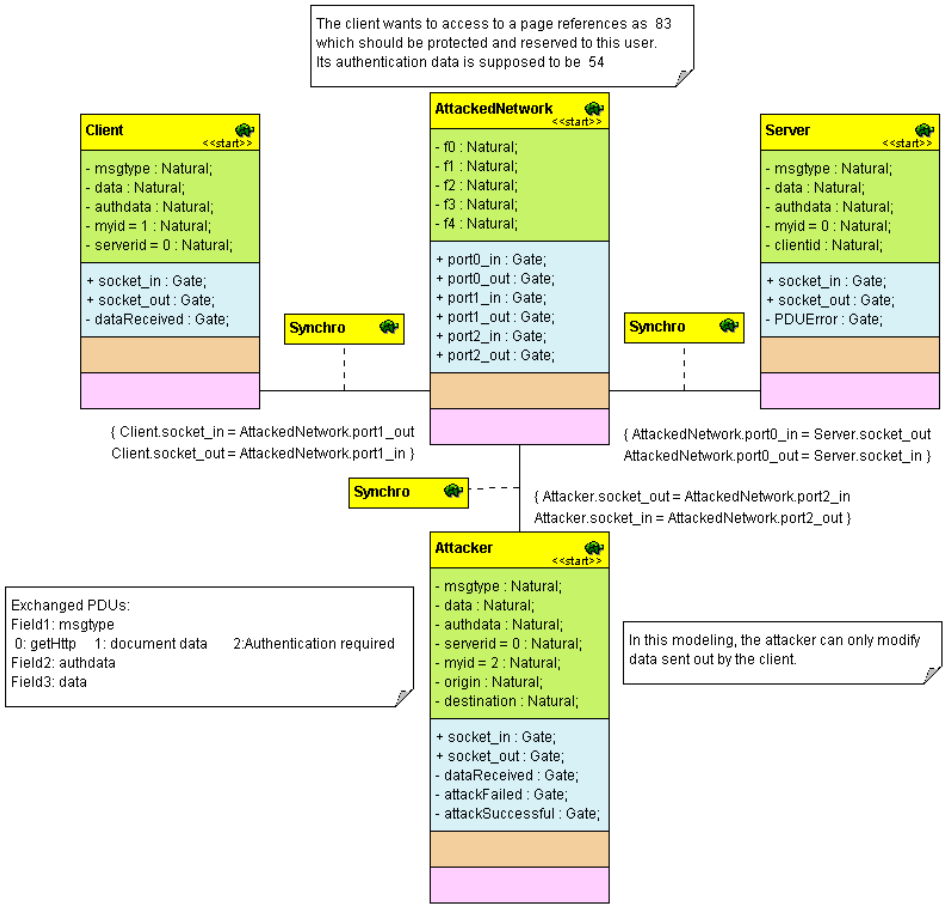

Figure 8. Diagramme de classes du système étudié / *Class diagram of the system under study*

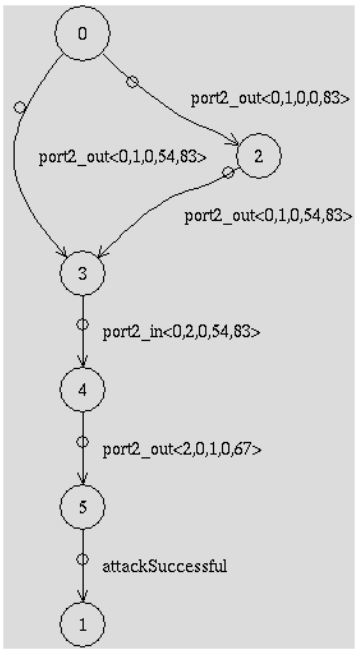

Figure 9. Graphe d'accessibilité réduit (minimisé) aux actions de l'attaquant / *Rechability graph minimized to the actions of the attacker*

Une fois cette validation effectuée, il est possible de générer du code Java à partir de cette conception TURTLE. Toutefois, ce code Java inclut celui du client, du serveur, de l'attaquant et enfin du réseau. Il est destiné à effectuer les simulations locales : ce code n'est absolument pas conçu sous la forme d'entités pouvant être déployées sur un réseau, car il est optimisé pour fonctionner comme un seul paquetage logiciel. De plus, ce code contient du code relatif à l'équipement réseau, ce qui n'est pas réaliste du point de vue logiciel. D'où l'intérêt d'organiser cette conception sous la forme d'un système distribué constitué de composants déployés et dont les communications réseau sont explicitement définies.

3. Déploiement du système

L'intérêt de réaliser le déploiement est, d'une part, de valider formellement le comportement du client et du serveur avec des liens asynchrones plus proches des liens réels (et non avec de simples synchronisations) et, d'autre part, d'obtenir des composants logiciels « client » et « serveur » à partir desquels des expérimentations vont pouvoir être menées.

Le déploiement est modélisé comme suit : sous TTool, nous créons un tcomposant « PkgClient » et un tcomposant « PkgServer » pour y copier respectivement les classes « Client » et « Server ». Nous créons un diagramme de déploiement composé de trois nœuds dédiés au serveur et aux deux clients (on voit que les liens du réseau peuvent être connectés sur la même porte destination). Sur les noeuds clients, nous déployons des instances du tcomposant « PkgClient ». Sur le noeud du serveur, nous déployons le tcomposant « PkgServer ». Dans cette étape de déploiement, l'attaquant est ainsi omis : celui-ci n'a été utilisé qu'à des fins de preuve formelle au niveau conception du système, la vérification à cette étape se fait en utilisant un outil d'analyse réseau et un forgeur de paquets HTTP.

Après avoir effectué une étape de validation formelle non détaillée ici, nous avons généré un code Java prévu pour fonctionner sur le composant matériel stipulé (PC). Nous avons pu tracer les échanges de données entre les clients et le serveur au moyen d'un analyseur de traces réseau [6]. En construisant le « bon » paquet depuis celui qui a été intercepté, i.e. le même paquet avec une adresse de retour différente, il est possible d'obtenir les données protégées par authentification. Cela confirme les résultats de la phase de validation formelle.

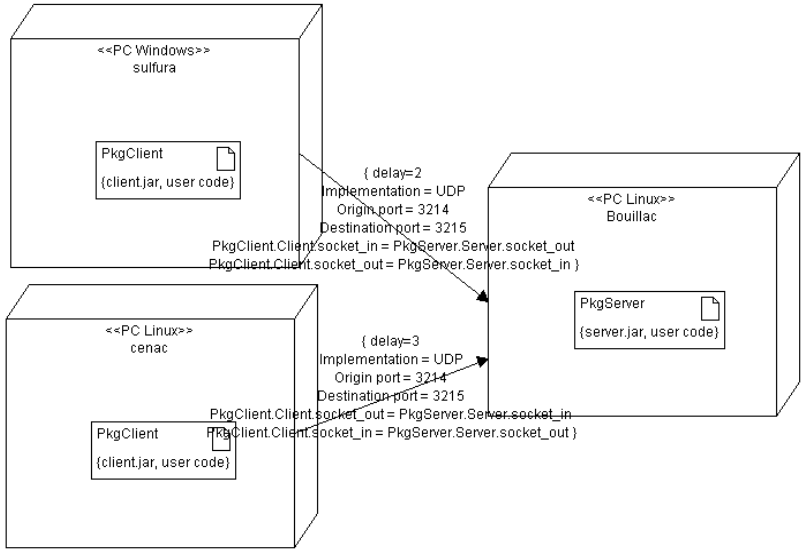

Figure 10. Diagramme de déploiement du système étudié / *The system's deployment diagram*

VII. DISCUSSION

Traiter un système avec le profil TURTLE complet (analyse, conception, déploiement) nous a permis d'identifier certaines limitations que nous allons détailler par la suite, et pour lesquelles nous allons fournir quelques éléments de solution.

La première limitation est d'ordre méthodologique. Si le passage de l'analyse à la conception est automatisé, le passage de la conception au déploiement ne l'est pas et soulève certaines interrogations notamment sur la notion de

liens asynchrones. En effet, la conception TURTLE est basée sur des relations de communication synchrones entre classes. Les composants déployés sont des groupes de classes de la conception. En raison de cette dualité synchrone / asynchrone, il est possible qu'un lien modélisé comme « synchrone » lors de la conception devienne asynchrone lors d'un déploiement, avec deux conséquences importantes. D'une part, les résultats de validation formelle obtenus en conception ne sont plus valables lors de la phase de déploiement. D'autre part, le double échange de données n'étant pas possible sur des liens asynchrones, des situations d'interblocage peuvent apparaître lors d'émission / réception simultanée de données sur des portes de communication. Bien entendu, par utilisation de multiples synchronisations, il est possible de modéliser des comportements de communication asynchrones lors de la conception, et de plaquer ces communications asynchrones sur les liens asynchrones des diagrammes de déploiement. En supposant que l'on sache identifier les sémantiques asynchrones utilisées en conception, on pourrait alors imaginer d'exploiter ces sémantiques d'asynchronisme pendant le déploiement pour faciliter la validation formelle. Il serait également souhaitable de pouvoir guider la génération de code et de sélectionner des protocoles de communication assurant les asynchronismes modélisés afin que le code exécutable généré possède la même sémantique que le code formel sur lequel des preuves ont pu être réalisées. Par exemple, si l'on considère les liens asynchrones des diagrammes de déploiement, l'utilisateur de TTool peut générer du code pour différents protocoles de communication (UDP, TCP, RMI), mais sans aucune garantie que le protocole et les bibliothèques logicielles utilisés par le générateur de code respecteront la sémantique asynchrone utilisée lors de la validation formelle.
Finalement, les solutions à ces difficultés ne sont pas triviales. Nos idées pour résoudre en partie ces problèmes sont les suivantes :

- Ajouter un opérateur d'asynchronisme aux diagrammes de conception ;
- Guider le déploiement en interdisant (ou en signalant ouvertement) le déploiement d'un lien synchrone sur un lien asynchrone, ou d'un lien asynchrone vers un lien asynchrone de sémantique différente (mauvais choix du protocole sur le lien). On pourrait imaginer des conseils du style « attention, vous avez plaqué un lien asynchrone sur un lien RMI qui ne possède pas la même sémantique ».
- Enfin, développer une bibliothèque RT-LOTOS représentant les différentes sémantiques associées aux liens UDP, TCP ou RMI, et effectuer la validation formelle des diagrammes de déploiement en utilisant ces bibliothèques.

La deuxième limitation concerne la génération de code.
En effet, il est parfois difficile de générer du code dans certaines conditions, notamment dans les cas d'opérateurs non déterministes. En effet, les traductions en Java des délais non déterministes et des choix non déterministes ne sont pas triviales.
Dans le cas du délai non-déterministe, le traducteur actuel tire au hasard une durée dans l'intervalle possible, et se met en attente pour cette durée. Cette sémantique n'est bien entendu pas la bonne lorsque le délai a été utilisé pour modéliser le temps de traitement d'un algorithme.
Dans le cas d'un choix non-déterministe, nous avons été confrontés au problème de la traduction de la sémantique donnée en RT-LOTOS vers une sémantique équivalente en Java. Cette sémantique RT-LOTOS est la suivante : lorsqu'un choix non déterministe est rencontré, toutes les actions accessibles depuis les différentes branches sont *sensibilisées*, c'est à dire qu'elle sont offertes en synchronisation. Lorsqu'une de ces actions peut être exécutée, le choix est réalisé, c'est à dire que la branche du choix est définitivement choisie. Considérons l'exemple représenté à la Figure 11. Les actions de T1 ne sont pas synchronisées avec un processus extérieur, c'est à dire que dès qu'elles sont sensibilisées, elles peuvent être exécutées. Au niveau du choix, trois actions sont sensibilisées : *action1*, *action2* et *action3*. Si *action1* est exécutée, alors action3 est désensibilisée et *actions2* est exécutée. Si *action2* est exécutée, alors *action3* est désensibilisée et *action1* est exécutée. Enfin, si action3 est exécutée, *action1* et *action2* sont désensibilisées.

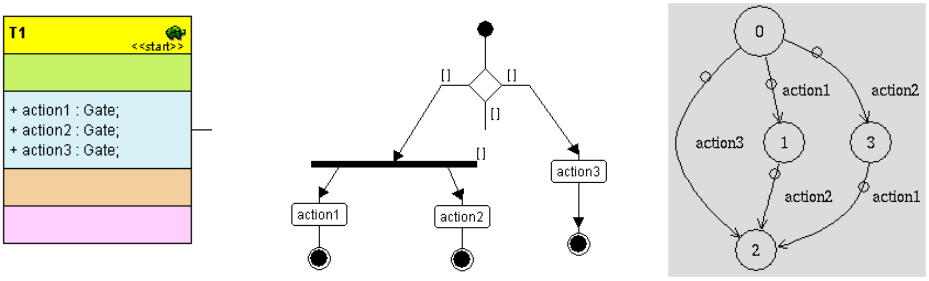

Figure 11. Une conception TURTLE et le graphe d'accessibilité lui correspondant / *A TURTLE design, and its corresponding reachability graph*

Traduire cette sémantique en Java est particulièrement complexe. En effet, il faut introduire la notion de sensibilisation d'actions, sensibiliser toutes les actions depuis le choix, et dès qu'une action est sensibilisée, opter pour la branche du choix. Les langages de programmation tels que Java sont plutôt adaptés à modéliser des choix déterministes, i.e. dont les gardes sont explicitement définies et disjointes. Lorsque cela n'est pas le cas, notre générateur de code tire pour l'instant au hasard une des branches. Cela a pour effet de forcer le choix d'une action sensibilisée, au détriment d'une liste d'actions. Si l'action choisie ne peut jamais être exécutée, notre programme Java se retrouve dans une situation de *deadlock* qui n'existait pas en phase de modélisation. En attendant une prochaine version de notre générateur de code, nous conseillons bien entendu de ne générer du code exécutable que pour des conceptions ou des déploiements dont les choix sont déterministes.

VIII. TRAVAUX DU DOMAINE

Les travaux concernant la validation des architectures distribuées se focalisent généralement sur la modélisation des protocoles de communication mis en œuvre dans ces architectures [23]. La modélisation des protocoles avec UML a été étudiée à diverses reprises depuis les prémices de la normalisation du langage UML à l'OMG. Depuis, plusieurs études ont été publiées, tout d'abord, sur des exemples académiques [7] [8], puis, sur des systèmes coopératifs [9], des systèmes de commerce électronique [10], des systèmes multi-agents [11] [12] [14] et enfin sur les approches ODP [13]. La réutilisation de composants de communication devient un enjeu important [15] [16]. Ces travaux s'appuient dans leur ensemble sur la version normalisée d'UML.

Les contributions sur UML pour les systèmes distribués proposent généralement des améliorations à UML basées sur les interactions entre les composants, et sur les contraintes de communication entre ces composants. Par exemple, AUML [17] introduit la notion de *Protocol Diagram* qui étend les diagrammes de séquence avec des opérateurs logiques afin d'exprimer de façon explicite la causalité, la synchronisation et la diffusion. Afin de décrire les contraintes de sécurité dans les systèmes distribués, [21] étend les diagrammes de déploiement UML à l'aide de stéréotypes associés aux liens entre noeuds. Notre approche reprend des éléments similaires à ces deux dernières approches, en proposant notamment en plus la validation formelle aux deux étapes. [19] propose aussi une approche formelle UML pour les systèmes distribués, approche qui s'appuie aussi sur les algèbres de processus. Par contre, le système distribué est décrit sous la forme d'interactions entre classes et sous la forme de machines à états, et non sous la forme d'un déploiement de composants. De plus, cette contribution ne propose pas de génération automatique de code, ni d'opérateurs temporels similaires à ceux de TURTLE. Une dernière approche consiste à modéliser le système distribué à des fins de performance et non de validation formelle. Ainsi, l'approche [20] consiste une fois encore à annoter les diagrammes UML avec des stéréotypes du domaine. Toutefois, cette contribution se focalise uniquement sur les diagrammes d'interactions UML 1.5, et leur transposition à UML 2.0 ne parait pas triviale.

Notons enfin que l'outil UML-AUT [18] supporte déjà les diagrammes de déploiement dans un environnement de conception de systèmes distribués mais nous n'y avons pas trouvé trace de formalisation de ces diagrammes.

IX. CONCLUSION

Dans le prolongement de [2], cet article a donné au profil UML TURTLE les ingrédients nécessaires au traitement de l'aspect « répartition » des systèmes bâtis en environnement Internet ou Middleware. Les contributions se situent au niveau des extensions apportées aux diagrammes de composants et de déploiement. Les premiers regroupent les objets définis dans le diagramme d'objets. Les seconds permettent de préciser les caractéristiques de la liaison Ethernet ou du bus logiciel. De plus, des algorithmes de génération de code applicables au profil TURTLE ainsi enrichi ont été définis et implantés dans l'outil TTool.

A ce jour, TTool permet d'éditer les diagrammes de composants et de déploiement, et de générer du code RT-LOTOS et Java. Notre ambition à court terme est de pouvoir modéliser plus précisément les composants matériels d'un système embarqué ou d'un système-sur-puce, et de pouvoir ajouter des composants TURTLE sur ces composants matériels. Par exemple, pouvoir ajouter des composants TURTLE sur le CPU d'un système-sur-puce ou sur un microcontrôleur, ou autre. Par ailleurs, nous souhaiterions que TTool puisse prendre en compte les contraintes de mobilité que l'on trouve dans bon nombre de systèmes embarqués et pouvoir valider formellement ces systèmes avant de générer les agents mobiles correspondants. Enfin, nous souhaitons travailler sur les relations conception TURTLE – déploiement TURTLE, afin de guider au mieux l'utilisateur du profil lors des validations formelles et de

la génération de code.

REFERENCES

[1] L. Apvrille, J.-P. Courtiat, C. Lohr, P de Saqui-Sannes , “TURTLE: A Real-Time UML Profile Supported by a Formal Validation Toolkit”, IEEE Transactions on Software Engineering, Vol. 30, No. 7, pp. 473-487, July 2004.

[2] L. Apvrille, P de Saqui-Sannes, F. Khendek, “TURTLE-P: un profil UML pour la validation d'architectures”, 10ème Colloque Francophone sur l'Ingénierie des Protocoles (CFIP'2003), Paris (France), 7-10 Octobre 2003, Hermès, pp.17-32.

[3] L. Apvrille, P. de Saqui-Sannes, F. Khendek, “Synthèse d'une conception UML temps-réel à partir de diagrammes de séquences", 11ème Colloque Francophone sur l'Ingénierie des Protocoles (CFIP'05), Bordeaux, France, 2005, Hermès.

[4] TTool, http://labsoc.comelec.enst.fr/turtle

[5] UML 2.0 Infrastructure Specification, http://www.omg.org/docs/ptc/03-09-15.pdf

[6] Ethereal. http://www.ethereal.com/

[7] C. Jard, J.-M. Jézéquel, F. Pennaneach, “Vers l’utilisation d’outils de validation de protocoles dans UML”, Techniques et Sciences Informatiques, v.15, n°11, pp. 1-15, 1998.

[8] M. Jaragh, K.A. Saleh, “Modeling Communications protocols using the Unified Modeling Language”, TENCON’2000, Intelligent Systems and Technologies for the New Millenium, Kuala Lumpur, Malaysia, September 2000.

[9]] J. M. M. Espinosa, O. Nabuco, K. Drira, “A UML Model for Session Management in Collaborative Design for Space Activities”, 8th European Concurrent Engineering Conference (ECEC'2001), Valence, Spain, pp.170-174, April 2001.

[10] I.W. Siu, Z .S. Guo, “The Secure Communication Protocol for Electronic Ticket Management System”, Univ. of Macau, June 2001

[11] K. Kavi, D.C. Kung, H. Bhaambhani, G. Pancholi, M. Kanikarla, R. Sah, “Extending UML to Modeling and Design of Multi-Agent Systems”; submitted to the International Conference on Software Engineering, 2003

[12] J. Lind, “Specifying Agent Interaction Protocols with Standard UML”, Second International Workshop on Agent-Oriented Software Engineering (AOSE-2001), LNCS 2222, Springer Verlag, Heidelberg, March, 2002

[13] M. Born, E. Holz, O. Kath, “A Method for the Design and Development of Distributed Applications Using UML”, TOOLS-Pacific 2000, Sydney, Australia

[14] M.-P. Huget, “Extending Agent UML Protocol Diagrams”, Agent Oriented Software Engineering (AOSE-02), Fausto Giunchiglia and James Odell and Gerhard Weiss (eds.), Bologna, Italy, July 2002.

[15] M. M. Kandé, S. Mazaher, O. Prnjat, L. Sacks, M. Vittig, “Applying UML to Design an Inter-Domain Service Management Application”, UML’98, Mulhouse, France, June 1998.

[16] E. Cariou, “ Spécification de composants de communication en UML”, OCM 2000, Nantes, France, mai 2000.

[17] J. Wei, S.C. Cheung, X. Wang, “Exploiting Automatic Analysis of E-Commerce Protocols”, 25th Annual Computer Software and Application Conference, Chicago, USA, October 2001.

[18] A. Le Guennec, “Génie logiciel et méthodes formelles avec UML : spécification, validation et génération de tests”, doctorat de l’Université de Rennes I, juin 2001.

[19] N. Kaveh, W. Emmerich, “Deadlock Detection in Distributed Object Systems”, Proceedings of the 8th European software engineering conference Vienna, Austria , pp. 44 – 51, 2001 .

[20] H. Gomaa, D. A. Menascé, “Design and performance modeling of component interconnection patterns for distributed software architectures”, 2nd Int. workshop on Software and performance Ottawa, Ont., Canada, pp: 117 – 126, 2000.

[21] J. Jürjens, “UMLsec: Extending UML for Secure Systems Development”, UML 2002, Dresden, Sept.Oct. 2002, LNCS

[22] RFC 2617, “HTTP Authentication: Basic and Digest Access Authentication”, June 1999.

[23] H. Gomaa, “Designing Concurrent, Distributed, and Real-Time Applications with UML”, Addison-Wesley Professional, Jan.2000.

[24] J.-P. Courtiat, C.A.S. Santos, C. Lohr, B. Outtaj, “Experience with RT-LOTOS, a Temporal Extension of the LOTOS Formal Description Technique”, Computer Communications, vol. 23, No. 12, 2000, p. 1104-1123

[25] Outil RTL, http://www.laas.fr/RT-LOTOS

[26] Outil CADP-Aldebaran, http://www.inrialpes.fr/vasy/cadp/