Raffinement de protocoles de communication par transformation de modèle

Pascal Andre, Antoine Guérin, Anthony Rozen, Alexandre Gicquel

Raffinement de protocoles de communication par transformation de modèle

ANDRE Pascal[1], GUERIN Antoine, ROZEN Anthony, GICQUEL Alexandre[2]

[1] LS2N CNRS UMR 6004 - Université de Nantes, France
pascal.andre@ls2n.fr
[2] Etudiants Master ALMA, Université de Nantes

Abstract

Logiciel et communication par les réseaux sont devenus incontournables dans les systèmes cyberphysiques et les systèmes réactifs. Capteurs, actionneurs et contrôle peuvent être distribués (exemple de l'IoT ou du pilotage d'ateliers de production). Par ailleurs, du point de vue du développement mais aussi de la maintenance des logiciels, on se doit d'être réactifs vis-à-vis des perturbations liées à de nouveaux besoins, des défaillances ou des contraintes techniques et proposer rapidement de nouvelles versions logicielles. A MSR 2019, nous avions proposé une piste basée sur l'ingénierie des modèles pour la vérification des propriétés et l'automatisation du développement par transformations successives (raffinement) pour raccourcir le cycle de développement et de maintenance. Dans cet article, nous nous focalisons sur le raffinement des communications entre objets Nous expérimentons d'abord divers protocoles sur notre étude de cas revue, une introspection nous permet de poser les bases d'une couche communication qui nous permette de nous abstraire des implantations de protocoles de communication. Nous pouvons alors envisager une traduction assistée des modèles vers les plateformes cibles via une transformation de modèle d'envoi de message. Pour réaliser les expérimentations, les programmes sont déployés sur le robot Lego Mindstorm EV3 et pour Android. Cette approche peut être appliquée dans d'autres contextes.
Mots-clés: Ingénierie des modèles, Raffinement de communications, Messages, Transformation de modèles, Vérification

1 Introduction

Les logiciels et les communications par réseaux prennent une place prépondérante dans les systèmes automatisés, avec des systèmes toujours plus distribués et inter-connectés. Cela s'explique par la multiplication d'appareils mobiles ou distants (smartphones, Internet des objets), des réseaux de communication généralisés (fibre optique, 5G) ou des calculs à distance (Cloud, IA). Un enjeu majeur est de mettre à jour ces systèmes automatisés pour faire face à des changements d'exigences ou aux défaillances techniques qui conduisent à reconfigurer les systèmes matériels et logiciels. Les cycles de vie habituels du développement logiciel manquent de réactivité pour faire face à ces changements, car c'est tout le processus qui peut être impacté, de la modélisation au codage, en passant par les nécessaires vérifications de propriétés des systèmes. Dans [2], nous avons proposé un axe de recherche qui consiste à mettre en place une approche basée sur les modèles (Model Engineering) [8] pour construire un logiciel permettant la génération de code à partir de modèles, par transformation de modèle. Nous avons proposé un processus de développement général et nous nous sommes concentrés sur la transformation des diagrammes d'état (UML) en programmes Java. Pour illustrer l'approche, nous travaillons avec des robots Lego Mindstorm EV3 et des mobiles sur Android. Ils sont représentatifs des problématiques rencontrées pour les systèmes distribués réactifs.

Dans cet article, les motivations restent les mêmes que dans [2] et peuvent être résumées par *aider les développeurs de logiciels à concevoir et mettre en œuvre des applications distribuées de confiance.* Déléguer le codage systématique à la génération de code permet au développeur de se concentrer sur des

activités d'ingénierie plus élaborées et permet de réduire les exigences de test (une fois la transformation automatique validée). Nous nous plaçons dans une vision *Model Driven Development* (MDD) qui peut se résumer par *utiliser des abstractions pour mieux vérifier les systèmes logiciels et générer des parties du code source cible.* Nous avons structuré le processus de développement général dans [4] en insistant sur (i) le besoin de remonter des abstractions des frameworks logiciels utilisés dans des modèles de plate-forme (*Platform Dependent Model* - PDM) car le raffinement pur relève de l'ingénierie et ne peut être automatisé et (ii) la nécessité de définir des transformations de modèles systématiques. Ici, nous continuons sur cette ligne en nous concentrant sur l'aspect "communications". Au niveau abstrait (modèle logique), la distribution est implicite et les objets échangent par envoi de message ou de signaux. Au niveau de la mise en œuvre, l'envoi de message peut être un appel de méthode, un appel distant ou un appel de service réseau. La question de recherche est de savoir comment raffiner (automatiquement) le message envoyé en solutions opérationnelles ?

La réponse n'est pas simple car la distance sémantique peut être importante, notamment lorsque les objets communiquent selon des protocoles hétérogènes. Notre méthodologie de travail est structurée en quatre étapes (i) expérimentation empirique de la conception manuelle sur une étude de cas représentative, (ii) analyse des enseignements pour identifier le périmètre et les problèmes à résoudre pour automatiser le processus MDD, (iii) proposition d'une couche primitive de communication (PDM communication), (iv) transformation de modèle pour raffiner l'envoi de message abstrait en implémentations concrètes. Chaque résultat d'étape représente une contribution à la question de recherche de cet article, où nous nous concentrons plus sur des questions méthodologiques que techniques. Nous expérimentons d'abord divers protocoles sur notre étude de cas de guidage, desquels nous tirons des principes pour produire une première couche d'abstraction pour les communications qui permet une transformation de modèle d'envoi de message. Cette approche peut être appliquée dans d'autres contextes.

L'article est organisé comme suit. Nous commençons par présenter le contexte dans la section 2, nous révisons aussi le contexte de l'étude de cas et soulignons les caractéristiques de communication. Nous illustrons les expériences de mise en œuvre manuelle de l'étude de cas dans la section 3. L'analyse et l'introspection de ces expériences sont synthétisées dans la Section 4 et la partie spécification de primitives dans la Section 5. Nous ajoutons le volet communication dans le processus de transformation de modèles en section 6 Enfin, dans la Section 7, nous faisons une comparaison avec les différents travaux plus ou moins connexes avant de conclure.

2 Contexte

L'approche MDD est un paradigme de développement qui utilise des modèles comme artefact principal du processus de développement. Habituellement, dans MDD, l'implémentation est (semi)automatiquement générée à partir des modèles [8]. Le processus unifié à deux voies (2TUP) [17] illustre une itération de développement logiciel dans l'esprit de MDD : le modèle d'analyse immergé dans un environnement technique sera appelé un modèle de conception, comme illustré par dans la Figure 1 de [2]. Le 2TUP met en évidence le rôle de la conception logicielle qui fait correspondre le modèle logique (issu de l'analyse des besoins) avec le modèle technique (issu d'une analyse technique) pour apporter une solution implantable. Les modèles peuvent être écrits à l'aide de langages de modélisation généraux tels que UML [5], SysML [11] ou de langages spécifiques *Domain Specific Languages* (DSL).

Dans des travaux précédents [2, 3], nous avons comparé différentes approches pour automatiser 2TUP : développement manuel, génération de code depuis les modèles, transformations pas à pas. Cette dernière est privilégiée du fait de sa réplicabilité et de son paramétrage et nous avons proposé un processus de conception générique dans [4] qui met l'accent sur le rôle du modèle dépendant de la plate-forme (PDM), l'abstraction des cadres techniques. Nous y avons aussi souligné la nécessité de définir de manière systématique les transformations indépendemment de leur implantation.

Dans la vision MDA, un processus de transformation est une séquence de transformations permettant de passer d'un *Platform Independent Model* (PIM) à un *Platform Specific Model* (PSM) plus concret. Les modèles logiques utilisés en entrée du processus font abstraction de l'environnement technique et des exigences non fonctionnelles. La conception consiste à combiner le modèle logique avec l'infrastructure technique pour obtenir un modèle exécutable. La génération de code elle-même n'est pas concevable en une seule étape de transformation, en raison de la distance sémantique entre le modèle logique et la cible technique, composée d'aspects orthogonaux mais corrélés, appelés domaines sur lesquels le modèle initial doit être combiné.

Travailler avec des transformations simples réduit les problèmes de cohérence et de complétude des transformations. Sur la base de ces considérations, nous adoptons un principe que nous qualifions de "transformations par petits pas". La complexité n'est pas dans la transformation individuelle mais dans le processus de transformation. Une transformation complexe est composée hiérarchiquement d'autres transformations, jusqu'aux transformations élémentaires. Ces macro-transformations utilisent des informations de configuration. Évidemment, si le modèle de départ comprend un diagramme de composants et un diagramme de déploiement en UML, la transformation sera simplifiée. Il est à noter que tous les paramètres et décisions de la transformation doivent être conservés afin de rejouer le processus de transformation si le modèle initial change.

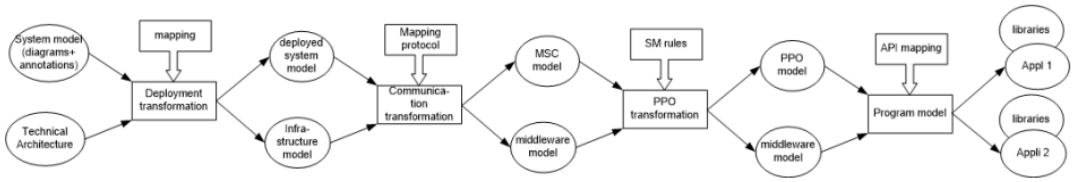

Figure 1: Processus général de transformation [2]

Le processus de la figure 1 est abstrait mais générique. Nous définissons quatre macro-transformations, chacune se concentrant sur un aspect de la conception : distribution (T1), communication (T2), modularisation (T3) et codage (T4). Un processus ne peut être automatisé que si tous les rouages sont connus avec précision. La pratique nous a montré que les transformations sont efficaces lorsque les modèles sont sémantiquement proches (par exemple, diagramme de classes et modèle relationnel pour la persistance de données - vision CRUD). Les transformations sont décrites par des règles de transformation ou sous forme algorithmique.

Dans les travaux précédents [2, 3] nous avons également expérimenté des transformations pour les modèles de classe UML et les Statecharts vers la programmation orientée objet (c'est-à-dire Java pour les études de cas). Nous avons souligné que la transformation des communications comme un problème ouvert, et nous nous le traitons maintenant. Une piste principale à creuser est la réalisation de l'envoi de messages UML.

À première vue, on pourrait penser que l'envoi de message de transformation (UML) est simple à mettre en œuvre par un appel d'opération. Mais les problèmes se posent dès qu'il existe une distribution sur plusieurs appareils distants car les envois de messages impliquent l'usage de protocoles de communication distante, qui par ailleurs peuvent avoir différentes implantations selon les *frameworks* ou les systèmes d'exploitations utilisés. En effet, ces protocoles ne fonctionnent pas de la même manière et n'ont pas les mêmes contraintes, par exemple dans un protocole WiFi, un serveur peut accepter plusieurs clients, alors que dans un protocole Bluetooth, un client et un serveur ne peuvent accepter qu'une seule connexion. De plus, la mise en œuvre d'un protocole de communication n'est pas la même d'un système d'exploitation à l'autre.

Etude de cas revue

Dans [2], l'étude de cas était une porte de garage, actionnée par une télécommande. Le prototype de la porte automatique a été implanté par une brique EV3 sous Lejos (système basé sur Linux) et la télécommande était une application Android pour mobile (smartphone ou tablette).

Dans l'étude de cas actuelle, nous avons remplacé la porte de garage par un portail extérieur selon un cahier des charges similaire. Afin de se concentrer sur les problèmes de communication, nous avons ajouté un véhicule (*RileyRover*) également contrôlé par une application IHM à distance sur mobile. Ce véhicule doit communiquer avec le portail comme illustré par Figure 2. Le nouveau système est

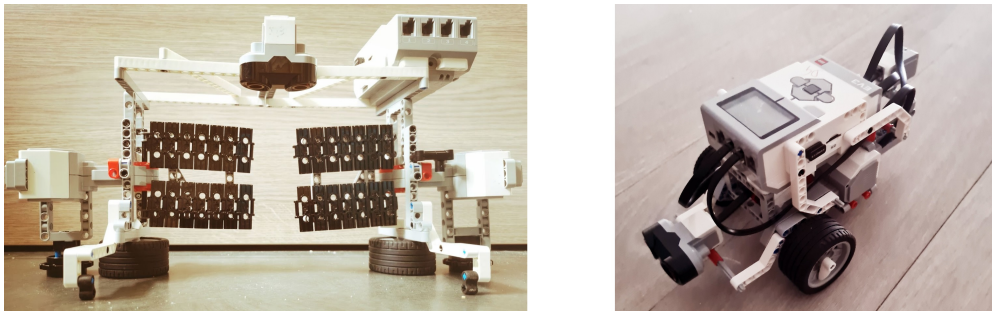

Figure 2: Maquette du portail et du véhicule

donc composé de quatre appareils : deux briques EV3 et deux applications Android avec deux appareils mobiles. Le système augmente en fonctionnalités mais aussi en contraintes. En particulier, un seul canal Bluetooth est disponible par appareil EV3: il sera utilisé pour relier les télécommandes aux briques (choix de conception). Cela implique que la communication entre EV3 se fera via une connexion WiFi. Bien que de taille limitée, ce cas est représentatif des problèmes rencontrés pour un système distribué avec des protocoles de communication hétérogènes. Les documents de spécification de cette étude de cas sont disponibles en ligne[1].

L'objectif de la mise en œuvre de cette étude de cas est de comprendre comment implanter les interactions entre objets distants sachant que dans le modèle UML ce sont des envois de messages. Trois cas sont traités : (i) communications entre un portail et un véhicule (wifi) : le portail accepte les demandes lorsqu'il reçoit la bonne adresse IP d'un véhicule, (ii) communications entre une application Android et un portail pour actionner les ouvertures/fermeture à distance, (iii) communications entre une application Android et un véhicule pour le pilotage à distance du véhicule.

Une évolution serait de traiter la communication entre deux véhicules, par exemple pour éviter une collision. L'étude de cas implique une communication entre télécommandes, portails et véhicules, simulant d'un système domotique réel. L'ensemble du système fonctionne comme suit. Le portail protège l'accès à la propriété, il est initialement fermé. Lorsqu'un véhicule s'approche du portail, une connexion est établie et le portail interroge le véhicule puis vérifie s'il est autorisé à passer. Dans l'affimative, le portail s'ouvre. Une fois le véhicule passé, il se ferme.

Selon les principes donnés dans [2, 4], les modèles d'entrée doivent être expressifs car (i) 1) la transformation de modèle peut inférer de nouveaux éléments mais ne peut pas les imaginer et (ii) la qualité des entrées influence la qualité des sorties pour le raffinement de modèles. Nous supposons donc une vérification sur les modèles d'entrée de propriétés telles que la complétude et la cohérence. Ce sujet sort du cadre immédiat de cet article mais le lecteur trouvera des pistes dans [1]. Dans la suite, nous implantons les programmes (transformation manuelle) pour lever les problèmes.

[1] https://ev3.univ-nantes.fr/projets-ev3/

3 Conception et Implantation des Communications

Le point de départ est une application[2] pour le véhicule et le portail. Chaque application constitue un sous-système indépendant (portail, véhicule) que nous allons intégrer. Chaque sous-système est lui-même composé de deux applications, une pour l'appareil EV3 et une pour le contrôle IHM sous Android. Ces applications distantes s'échangent par des connexions Bluetooth dans chaque cas. Elles n'ont pas été conçues explicitement pour être composés à un niveau supérieur, nous devrons donc les modifier, notamment pour l'intégration. Cette section décrit l'expérimentation de développement [12].

3.1 Connexion/communication entre le portail et la télécommande

Le portail est contrôlé à distance par une application Android qui émule la télécommande. Pour initialiser la connexion, le portail gère simplement la réception Bluetooth des données. Tant que la télécommande ne demande pas de se connecter, le portail attend. Au lancement de l'application Android, le test de connexion est lancé. Pour établir cette connexion, il est nécessaire que chacun des appareils ait des paramètres différents. Figure 3 décrit les principales classes mais on trouvera dans [2], une version abstraite des modèles d'entrée. Le portail est assimilé à un serveur, il attend une requête de connexion

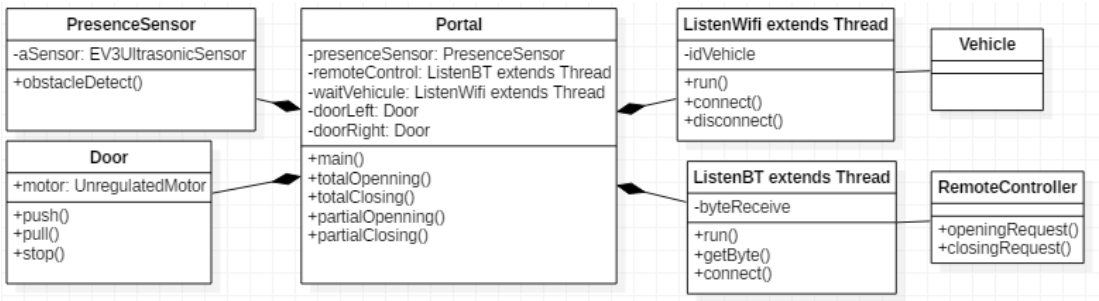

Figure 3: Diagramme de classe - Portail

avec comme paramètres le mode de connexion, ainsi que le temps maximum d'attente d'une connexion. La télécommande est un client, les paramètres sont le nom du serveur et le mode de connexion. Les différents modes de connexion sont : (i) NXTConnection.RAW (mobile Android) (ii) NXTConnection.PACKET (brique EV3) (iii) NXTConnection.LCP (menus de la brique) Un scénario est donné en exemple dans Figure 4. Une fois connecté, le portail exécute une boucle pour recevoir les commandes par bluetooth et les exécuter. La plupart des étapes ci-dessus peuvent être automatisées. Cependant, certains paramètres doivent encore être fournis au convertisseur par le développeur. L'application mobile de la télécommande suit une architecture Android App. La plupart de ces fonctionnalités sont appelées dans la classe MainActivity et la classe ConnectionBluetoothActivity. Lorsque l'utilisateur déclenche un événement, le système lance l'activité correspondante.

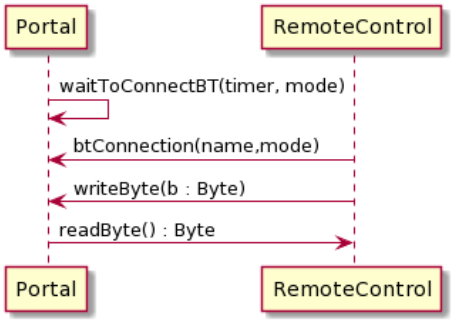

Figure 4: Interaction Portail/Télécommande

[2]Plus précisément nous avons à disposition différentes implantations implémentées préalablement par différents groupes d'étudiants, par exemple certains portails ont deux battants et d'autres un seul.

3.2 Connexion/communication entre le véhicule et la commande mobile

L'architecture est similaire à celle du portail. Figure 5 résume les classes impliquées.

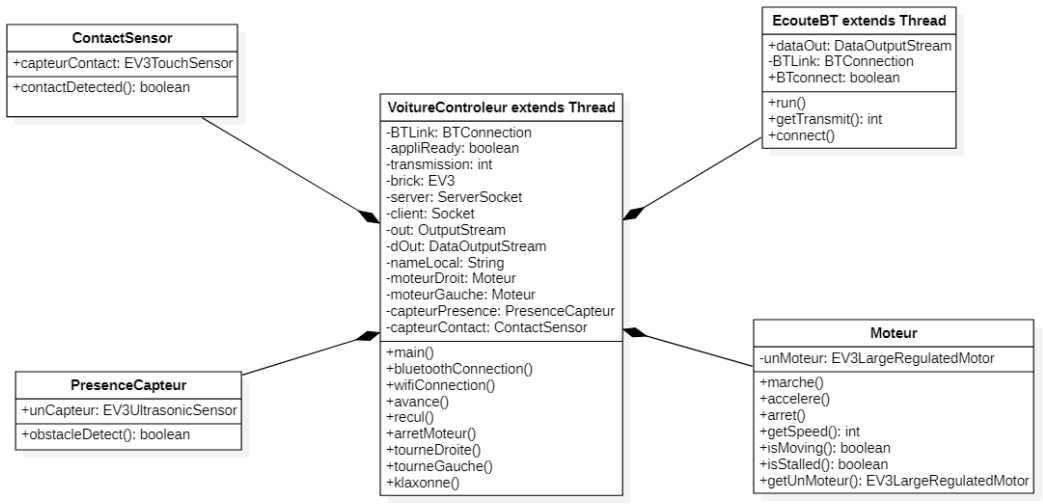

Figure 5: Diagramme de classe - Véhicule

La seule différence est le nombre d'actions possibles. L'IHM propose des mouvements avant-arrière, des freins-arrêts, des commandes de vitesse ou des actions de connexion. Pour la communication, le scénario est similaire à celui de la Figure 4.

3.3 Connexion/communication entre le véhicule et le portail

Le véhicule et le portail échangent des informations à distance via une connexion WiFi, où le portail est le client et le véhicule est le serveur (*cf.* Figure 4). On notera le parallélisme d'écoute des événements lié aux connexions WiFi et Bluetooth. Lorsque le portail détecte le véhicule, le test de connexion est lancé. Du côté du véhicule, un seul port de connexion est requis en paramètre. Du côté du portail, on a le même port de connexion et l'adresse IP du serveur. Une fois connecté, le véhicule demande à entrer. Si le portail reconnaît ce véhicule alors il ouvre les portes. Quelques secondes après le passage du véhicule, le portail se ferme et se déconnecte du véhicule. Ces expérimentations ont permis d'identifier des problèmes techniques tels que l'accès unique Bluetooth, la connexion Wifi par client/serveur et de déterminer les paramètres de configuration par essais successifs. La connexion sans fil a été implémentée en Java par des sockets (Socket et SocketServer) et des threads implantent le parallélisme. L'étape suivante consiste à approfondir l'étude des informations de communication afin de généraliser l'envoi de message à plusieurs protocoles de communication.

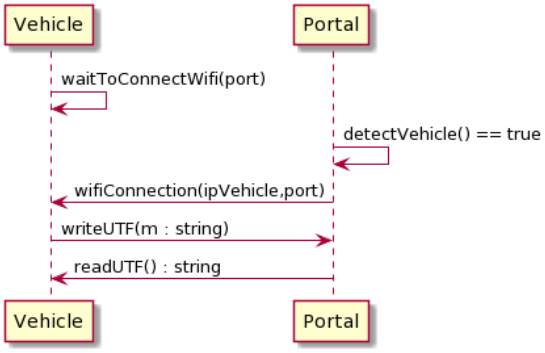

Figure 6: Interaction Portail/Véhicule

4 Introspection du code pour les Connections/Communications

Lors de la transformation manuelle (*cf.* section 3), nous avons géré deux types de protocoles sans fil (WiFi et Bluetooth). Pour tracer les échanges de communications, on peut observer le comportement sur chaque acteur (projections *e.g.* lignes de vie des diagrammes de séquence de figure 4) ou intercepter les communications sur le support de communication. La première solution est délicate lorsqu'il y a des pertes de message ou que le processeur est un système embarqué, comme l'EV3.

4.1 Protocole WiFi

Pour capturer les paquets réseau envoyés par une connexion WiFi, on peut utiliser le logiciel Wireshark[3]. Cependant, ce logiciel n'est utilisable simplement que sur un ordinateur. Pour observer les échanges entre EV3, nous les avons interceptés par un ordinateur, qui filtre les paquets TCP comme illustré par figure 7.

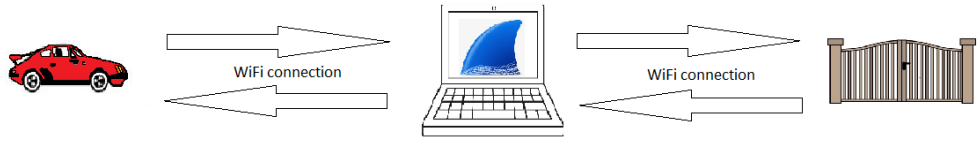

Figure 7: Interception des transmissions WiFi

Nous observons les interactions séparément. La Figure 8 est un extrait des communications. Le tableau du haut (resp. bas) représente les paquets du véhicule[ordinateur] -> portail (resp. véhicule -> [ordinateur] portail) et la figure 9 les représente sous forme de scénario (*Message Sequence Chart* - MSC). Les paquets `SYN` sont des demandes de synchronisation ou de connexion TCP. Les paquets `ACK` sont des accusés de réception des envois précédents. Les paquets `PSH` sont des envois de données (5 octets).

No.	Time	Source	Destination	Protocol	Length	Info
20	5.742605	192.168.1.22	192.168.1.16	TCP	74	45437 → 1236 [SYN] Seq=0 Win=5840 Len=0 MSS=1460 SACK_PERM=1 TSval=4294951957 TSecr=0 WS=2
21	5.742834	192.168.1.16	192.168.1.22	TCP	66	1236 → 45437 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=1460 WS=256 SACK_PERM=1
22	5.765369	192.168.1.22	192.168.1.16	TCP	54	45437 → 1236 [ACK] Seq=1 Ack=1 Win=5840 Len=0
23	5.823410	192.168.1.16	192.168.1.22	TCP	59	1236 → 45437 [PSH, ACK] Seq=1 Ack=1 Win=131328 Len=5
24	5.863014	192.168.1.22	192.168.1.16	TCP	54	45437 → 1236 [ACK] Seq=1 Ack=6 Win=5840 Len=0

No.	Time	Source	Destination	Protocol	Length	Info
125	11.434720	192.168.1.16	192.168.1.22	TCP	66	52130 → 1234 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 WS=256 SACK_PERM=1
128	11.483888	192.168.1.22	192.168.1.16	TCP	66	1234 → 52130 [SYN, ACK] Seq=0 Ack=1 Win=5840 Len=0 MSS=1460 SACK_PERM=1 WS=2
129	11.484105	192.168.1.16	192.168.1.22	TCP	54	52130 → 1234 [ACK] Seq=1 Ack=1 Win=131328 Len=0
130	11.632187	192.168.1.22	192.168.1.16	TCP	59	1234 → 52130 [PSH, ACK] Seq=1 Ack=1 Win=5840 Len=5
131	11.682699	192.168.1.16	192.168.1.22	TCP	54	52130 → 1234 [ACK] Seq=1 Ack=6 Win=131328 Len=0

Figure 8: Capture des paquets WiFi

La procédure d'envoi des messages est la même dans les deux situations. Cependant, il existe quelques différences donc les champs suivants ont été ajoutés du côté vehicule[ordinateur] -> portail : 1) time stamp value `TSval`, 2) Timestamp Echo Reply `Tsecr`. Rappelons que le véhicule (resp. portail) représente le serveur (resp. client). Il y a aussi des champs optionnels : 1) `MSS` Maximum Segment Size (sets the largest segment that the local host will accept), 2) `WS` Window Scale and 3) `SACK_PERM` Selective ACK Permitted. Nous avons observé que 3 paquets étaient nécessaires pour établir une connexion (les trois premiers paquets) et que seulement 2 paquets étaient nécessaires pour envoyer une communication

[3] https://www.wireshark.org/

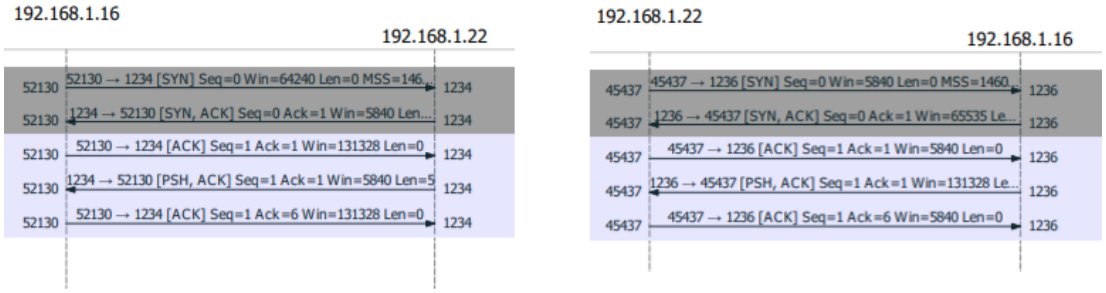

Figure 9: Scénario de la capture des paquets

(les deux derniers paquets). Nous avons également remarqué qu'il y avait différentes options pour une connexion TCP selon le système d'exploitation.

4.2 Protocole Bluetooth

Nous souhaitions faire de même pour les connexions Bluetooth mais le logiciel Wireshark n'intègre pas automatiquement une solution pour analyser les protocoles Bluetooth sous Windows[4]. Lors de l'installation de Wireshark, on sélectionne le package optionnel USBPcap pour capturer le trafic USB. La plupart des ordinateurs avec Bluetooth utilisent en interne le bus USB ou un dongle USB (EV3).

Un autre problème est l'unicité de la connexion Bluetooth sur de nombreux appareils qui empêchent d'utiliser un ordinateur comme intercepteur. Nous avons tenté avec l'émulateur d'Android Studio mais il n'inclut pas de logiciel virtuel pour Bluetooth[5]. Ces limitations empêchent le filtrage des communications BT par un ordinateur intercepteur. Seul un mobile peut jouer le rôle de télécommande et d'espion de communication. Nous essayons de capturer les paquets Bluetooth via une application indépendante sur le téléphone[6] , mais le résultat n'était pas exploitable. Une enquête plus approfondie est nécessaire[7].

Il est à noter que connaître le protocole est une chose mais sa mise en œuvre en est une autre. Bien qu'aucun détail ne soit fourni ici, nous avons observé des différences dans les paramètres primitifs pour Windows, Lejos, Linux et MacOS. L'étude d'introspection montre les interactions de bas niveau que nous allons abstraire dans une couche primitive afin de présenter des protocoles de communication uniformes pour les transformations d'envoi de message.

5 Couche Support de Communication

Dans cette section, nous nous concentrons sur l'abstraction des communications afin de présenter une interface uniforme qui correspond au modèle dépendant de la plate-forme (PDM) de l'ingénierie inverse dirigée par les modèles (*Model Driven Reverse Engineering* MDRE) des *frameworks* techniques [3].

L'interface définit un ensemble de primitives de communication qui fait abstraction des modes et conditions (synchrone ou asynchrone, sans état ou session, etc.) et des réseaux support (fil, WiFi, Bluetooth). Elle doit être compatible avec le modèle logique (*e.g.* envoi de message et signaux en UML) pour envisager la transformation de modèles.

[4] https://tewarid.github.io/2020/08/20/analyze-bluetooth-protocols-on-windows-using-wireshark.html
[5] https://developer.android.com/studio/run/emulator#wifi
[6] Bluetooth traffic sniffinghttps://profandroid.com/network/bluetooth/sniffing.html
[7] *e.g.* Packet Capture https://www.dz-techs.com/fr/wireshark-alternatives-for-android

5.1 Couches, Messages and Primitives

L'introspection nous a montré que les mêmes protocoles peuvent être implantés différemment, selon le système d'exploitation, sa version et le fournisseur de la bibliothèque de protocoles. Cela varie aussi avec le rôle (client/serveur) et le type d'envoi (synchrone, asynchrone, résultat, signal...). Pour masquer ces différences et éviter la duplication de code, nous avons conçu la couche de communication avec des classes abstraites et des interfaces spécialisées. La Figure 10 présente les classes principales de la couche de communication.

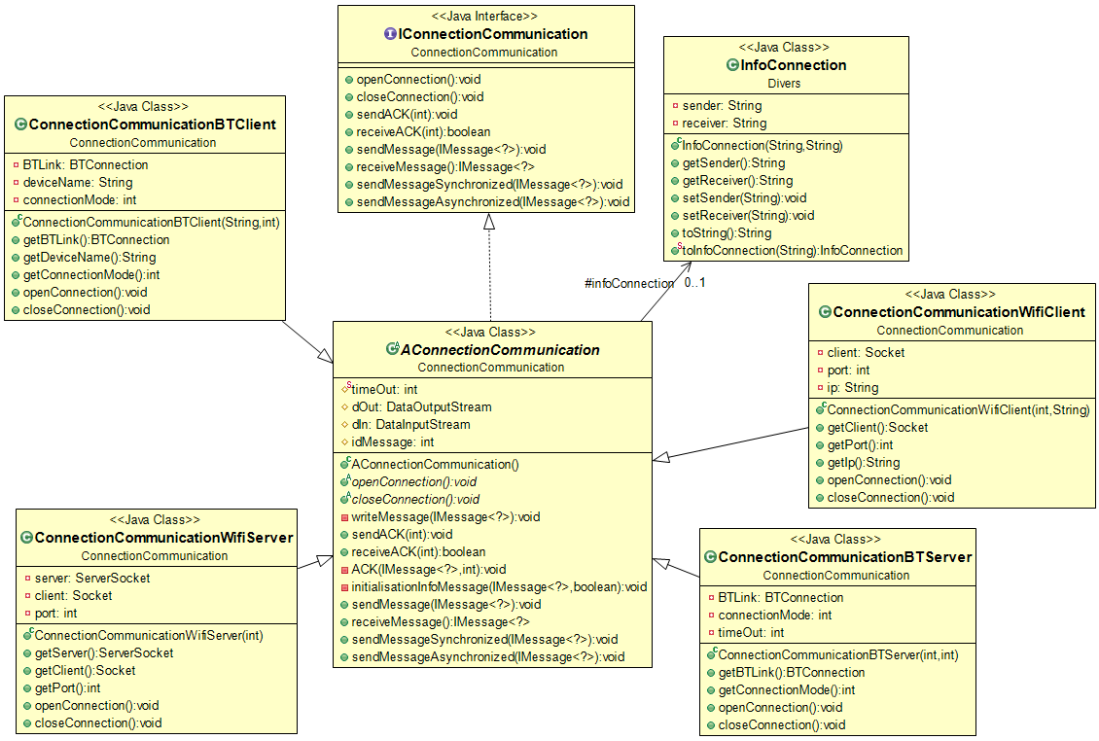

Figure 10: Classes de la couche communication

Pour rester le plus générique possible, un message envoyé ne doit pas dépendre du type du message envoyé, c'est le rôle de l'interface IMessage<T> et du type abstrait AMessage<T>. Ils décrivent le contenu d'un message et le comportement associé, par exemple la conversion de ou vers des chaînes. Un message peut accéder à InfoConnection *e.g.* expéditeur, destinataire et le InfoMessage, qui comprend l'identifiant du message, le type de message et si le message doit être reconnu. Le type de contenu du message évolue dans les sous-classes AMessage<T>. Une fabrique de messages permet de créer des instances d'un message selon le type indiqué dans une chaîne de message. Une classe d'encodage/décodage permet les conversions d'octets/chaînes. Les détails sont disponibles dans [12].

- Les services correspondent aux différents types de connexion possibles. Dans l'étude de cas, les implémentations sont prévues pour WiFi server, WiFi client, Bluetooth server LeJos, Bluetooth client LeJos and Bluetooth client Android.
- Les paramètres sont les données à fournir pour la configuration : 1) Wifi server (port), 2) Wifi client (port, IP), 3) BT server LeJos (mode, max waiting time), 4) Bluetooth client LeJos (device name, mode), 5) Bluetooth client Android (MAC).

- Les primitives correspondant aux ordres du protocole : 1) openConnection 2) closeConnection 3) sendMessage 4) sendMessageSynchronized 5) sendMessageAsynchronized 6) receiveMessage 7) sendACK 8) receiveACK

Les primitives de communication ci-dessus sont définies dans l'interface IConnectionCommunication et son implémentation abstraite AConnectionCommunication avec un timeOut (temps maximum avant d'envoyer un message si on n'a pas reçu d'accusé de réception), un flot d'entrée/sortie et l'identifiant de message suivant qui est simplement le nombre de messages. Plusieurs sous-classes concrètes implémentent les rôles client et serveur pour BT et WiFi. L'extension à la connexion filaire par exemple, le câble USB EV3 est similaire au cas WiFi. Notez que les paramètres de connexion sont donnés une seule fois à l'instanciation. La gestion du *timeout* est décrit dans la classe abstraite.

5.2 Conception et Implantation

Nous avons réalisé plusieurs types d'envoi de message : i) un envoi de message synchrone, le système attend un résultat pour pouvoir continuer, ii) deux envois de messages asynchrones (sans attente) selon qu'un résultat est attendu plus tard ou pas. Les primitives correspondant à ces envois de message et accusés de réception ont été ajoutées.

Dans les communications réseau, le raffinement vers le réseau physique ajoute des informations de service dans le contenu du message (*cf.* figure 11). Le corps du message est initialisé par l'utilisateur de la couche. Ensuite, la "Description du message" est renseignée avec l'identifiant du message de l'émetteur, receveur, le type du message et la demande d'un accusé de réception ou non.

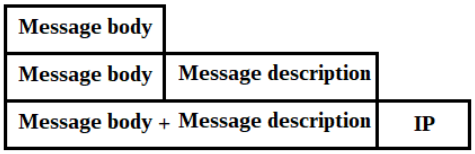

Figure 11: Abstraction de message

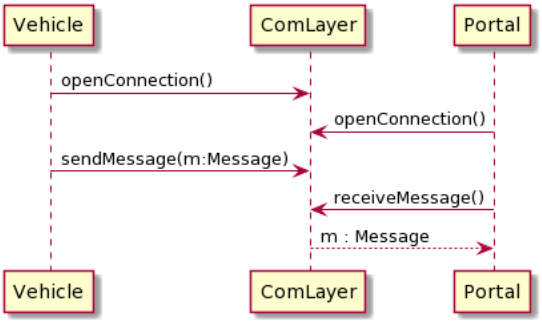

Figure 12: Interaction Portail/Véhicule raffiné

Ces informations sont définies à plus bas niveau selon le type de protocole utilisé. Le scénario de la Figure 4 est raffiné en utilisant la couche de communication dans la Figure 12, les communications de bas niveau dans la couche ComLayer. La couche a été implantée dans une archive Jar.

6 Application au processus de transformation

La dernière étape concerne les transformations de modèle. On commence par remplacer manuellement les parties de communication par appel primitif. Ensuite, nous complétons les règles transformations de modèle proposées dans [2, 4] sur le volet communication.

6.1 Réécriture des communications

La couche de communication a été intégrée à l'étude de cas (version 2) à des fins d'évaluation. Les classes de communication spécifiques ont été supprimées et leur appel API a été remplacé par des appels primitifs de couche de communication. À première vue, les primitives fonctionnent mais un vrai banc de test serait utile. L'implémentation du serveur et du client Bluetooth dans le framework LeJos, a due être refaite pour Android car non compatible. A la question "existe-t-il d'autres implémentations client/serveur Bluetooth qui sont différentes pour différents frameworks ?", la réponse est affirmative après quelques recherches. Par exemple, sur le site développeur Apple [8], on trouver les documentations concernant le framework Core Bluetooth d'Apple.

6.2 Processus de transformation de modèles revu.

La prise en compte des communications se fait à chaque étape du processus de la Figure 1:

- T1 Deployment transformation : le concepteur choisit la plate-forme de distribution : les nœuds avec leurs systèmes d'exploitation associés (*e.g.* dans notre cas il y avait deux briques EV3, deux appareils mobiles et capteurs/actionneurs) et les liens (*e.g.* câblé pour les capteurs et moteurs, sans fil pour les connexions mobile-EV3) avec les protocoles de communication associés aux liens (Bluetooth,WiFi, API...). Les informations de connexion (IP, port...) sont à fournir ici dans les fichiers de configuration.
- T2 Communication transformation : les actions de communication du modèle *e.g.* envois de messages, sont transformées en primitives de communication selon la configuration de distribution résultant de la transformation T1.
- T3 OOP transformation : les machines à états sont instrumentées selon les langages de programmation cibles, selon le nœud/OS de la transformation T1. Ce point était une contribution de [2].
- T4 Program transformation : le *mapping* de l'API raffine la primitive de communication vers les bibliothèques de communication. Les primitives de communication définissent des correspondances d'API implicites. Nous avons expérimenté une assistance pour rechercher des correspondances d'API candidats pour les communications filaires LeJos avec des capteurs et des moteurs en utilisant le pattern Adapter [4].

Les transformations bas niveau T3 et T4 ont été abordées dans [3, 4]. On s'intéresse ici à T1 et T2.

T1 (macro) Transformation Process

En entrée, le concepteur fournit un diagramme de déploiement UML où les classes sont réparties sur les nœuds. Des composants ou des paquetages peuvent structurer cette distribution. Nous supposons ici qu'il n'y a qu'un seul lien (et un seul protocole) entre deux nœuds[9]. Les liens sont renseignés avec protocole, client, serveur, adresses IP, mode session...

Pour chaque lien, deux classes proxy de communication sont générées, une pour chaque nœud ; ces proxys sont des sous-classes de AConnectionCommunication selon le protocole mentionné pour le lien entre les nœuds correspondants. Les liens de déploiement sont affinés par des associations entre les classes proxy.

Pour chaque classe C1 du nœud N1 qui déclare une action de communication à une instance d'une classe C2 du nœud N2, nous générons une association à la classe proxy qui implémente le lien entre N1 et N2 dans le schéma de déploiement. Les proxys peuvent inclure une table de routage qui relie les identifiants d'objet UML aux proxys de communication.

[8] https://developer.apple.com/documentation/corebluetooth

[9] [14], super-structure, p. 202 - le lien est un chemin de communication qui gère l'envoi du message.

T2 (macro) Transformation Process

Pour chaque paire de classe proxy de communication du modèle issu de T1, nous définissons une procédure d'initialisation qui configure la connexion physique au réseau. La couche primitive de communication est utile dans cette activité. De même, nous proposons des procédures de déconnexion pour fermer les sessions et les connexions.

Pour chaque envoi de message[10], nous analysons les fonctionnalités UML (sync/async, expéditeur , récepteur, paramètres, résultats...). Notons en particulier qu'un message UML est associé à une structure sendEvent-MessageEnd et à une structure receiveEvent MessageEnd-[11] qui peut être utilisé pour stocker les proxys. Ces fonctionnalités deviennent des arguments pour un message envoyé au proxy de communication, selon le service d'annuaire (de type LDAP)[12]. Un identifiant d'envoi de message transformé en une séquence d'envoi de message (niveau inférieur) et d'accusé de réception.

6.3 Vérification runtime

Comme mentionné dans la Section 2, les modèles UML d'entrée sont censés être vérifiés avant la transformation du modèle. Concernant les communications, on peut s'appuyer sur des travaux de transformations de modèles en automates synchronisés et ensuite de *model-checking* pour vérifier des propriétés de la dynamique (absence de deadlock par exemple) [1].

La vérification *runtime* concerne l'enrichissement des transformations pour la vérification à l'exécution des propriétés des modèles. L'idée est d'enrichir la couche de communication avec des fonctionnalités de journalisation pour stocker les informations de traçabilité. Ces informations permettent d'extraire les traces d'exécution et de vérifier les mêmes propriétés qui ont été vérifiées lors de la phase de modélisation. ici aussi on exploite les transformations de modèle pour extraire les informations et les écrire dans un format exploitable par des outils de *model-checking*.

7 Discussion

La génération de code à l'aide de MDA, le domaine reste inexploré en ce qui concerne un processus générique ; les 50 études principales de [19] se concentrent principalement sur des sujets spécifiques liés à des domaines spécifiques, aux systèmes embarqués, à la mobilité ou à la sécurité. A notre connaissance, la génération de code pour les communications en UML dans des systèmes distribués hétérogènes ouverts n'a pas été décrite dans la littérature (voir Tableau 1 dans [2]). Parmi les nombreux outils UML qui prennent en charge la génération de code[13], il est difficile de trouver lequel inclut vraiment l'envoi de message ; la solution intégrée génère un langage clair ou un middleware (homogène) par exemple Enterprise Service Bus (ESB).

Les outils autour d'Executable UML ou UML-RT (temps réel) se concentrent sur la description détaillée du comportement dynamique et prennent en charge la simulation [6, 20]. De telles approches prennent généralement en charge les machines à états et les langages d'action, *e.g.* le standard fUML/ALF, mais ne se concentrent pas sur les communications distribuées (intergiciel unique) et les spécificités des plates-formes cibles [7].

On peut expliquer ce déficit par la complexité du problème et notamment la distance sémantique entre les modèles et le code, qui font que la conception et la programmation restent des activités

[10] Il faut noter ici la grande complexité des règles de transformation car de nombreux concepts du méta-modèle UML sont impliqués, voir [14], super-structure chap. 11 Actions, chap. 13 Comportements courants, chap. 14 interactions, etc.

[11] [14], p. 478

[12] Une telle infrastructure améliore la traçabilité.

[13] Outils propriétaires tels que Sparx EA, Visual Paradigm, IBM Rational ou open source comme Papyrus ou Modelio.

d'ingénierie. Les approches *RoundTrip* [18] (symétrie entre le modèle et le code) ne sont pas suffisantes lorsque des machines à états ou un support en temps réel sont en jeu. La génération de code doit être réalisée par des transformations par étapes pour insérer en douceur les décisions de conception. Ciccozzi et al [10] ont montré que la transformation de modèles UML reste un problème difficile et que l'automatisation est plus adaptée à la simulation qu'au développement logiciel. De telles approches ont été implantées, par exemple, pour le *model-checking* et l'exécution via C++ [9] ou MoKa/Papyrus [13, 16]. L'interaction humaine dans les transformations reste prépondérante lorsqu'il existe des alternatives, comme le choix des protocoles de communication pour l'envoi de messages. Proposer une couche de communication, avec différentes implémentations concrètes, est conforme à l'idée de rétro-ingénierie des modèles de définition de plate-forme (PDM) qui facilitent la mise en œuvre (raffinement) des transformations de modèles [4].

Nous avons principalement abordé le raffinement des modèles logiques vers les programmes des applications par transformation de modèle. Mais les techniques peuvent également s'appliquer pour tester des modèles et générer des scénarios de test. En particulier, nous utilisons des diagrammes de séquence (*Message Sequence Charts*) pour décrire des scénarios de test à jouer sur l'application. Fondamentalement, les scénarios peuvent être utilisés pour tester des modèles par exemple, tester les machines à états UML et les événements de communication (envoi de messages, signaux). Grâce à la couche de communication (et aux fichiers de configuration), ils peuvent également être utilisés pour générer des cas de test au niveau de la programmation. Des informations à ce sujet peuvent être trouvées dans [15].

8 Conclusion

Dans [2] nous avons proposé une approche dirigée par les modèles pour aider les développeurs à concevoir des systèmes distribués réactifs à partir de modèles logiques écrits en UML par exemple. Dans cet article, nous nous sommes concentrés sur les aspects de la communication distante qui sont orthogonaux à la transformation des classes statiques et des machines à états dynamiques. Nous les avons d'abord implémentés manuellement dans une étude de cas étendue où les communications sans fil jouent un rôle important. Nous avons analysé les caractéristiques de communication afin de concevoir une couche primitive de communication qui sera un paramètre d'entrée pour la transformation de modèle, relative au raffinement vers code pour les envois de messages. Les idées développées ici peuvent être applicables à d'autres cas de raffinement de modèles de systèmes distribués réactifs.

Il reste beaucoup de travail à faire pour avoir un processus pleinement défini et outillé. La couche de primitives de communication mise en place permet des communications pair-à-pair. La couche primitive doit être validée par plus de tests et nous réfléchissons actuellement à une version *broadcast* avec MQTT (par exemple pour commander plusieurs portails ou véhicules en parallèle). La mise en œuvre de la transformation du modèle, au-delà de l'analyse de faisabilité déjà effectuée, y compris les communications et les machines d'état, est la principale perspective à moyen terme. Nous utilisons ATL. Enfin, la vérification des propriétés du modèle à l'exécution par analyse de traces rendra l'utilisation des modèles vraiment rentable en pratique. Les bénéfices réels d'une telle approche sont les gains affectés à l'évolution (maintenance) et à la réutilisation des logiciels.

References

[1] Pascal André, J. Christian Attiogbé, and Jean-Marie Mottu. Combining techniques to verify service-based components. In Luís Ferreira Pires, Slimane Hammoudi, and Bran Selic, editors, Proceedings of the 5th International Conference on Model-Driven Engineering and Software Development, MODELSWARD 2017, Porto, Portugal, February 19-21, 2017, pages 645–656. SciTePress, 2017.

[2] Pascal Andre and Yannis Le Bars. Conception assistée de contrôleurs d'automates depuis des modèles UML. In MSR 2019 - 12ème Colloque sur la Modélisation des Systèmes Réactifs, Nov 2019, Angers, France, Angers, France, November 2019.

[3] Pascal André and Mohammed El Amin Tebib. Refining automation system control with MDE. In Slimane Hammoudi, Luís Ferreira Pires, and Bran Selic, editors, Proceedings of MODELSWARD 2020, Valletta, Malta, February 25-27, 2020, pages 425–432. SCITEPRESS, 2020.

[4] Pascal André and Mohammed El Amin Tebib. More automation in model driven development. In J. Christian Attiogbé and Sadok Ben Yahia, editors, Model and Data Engineering - 10th International Conference, MEDI 2021, Tallinn, Estonia, June 21-23, 2021, Proceedings, volume 12732 of Lecture Notes in Computer Science, pages 75–83. Springer, 2021.

[5] Pascal André and Alain Vailly. GÉNIE LOGICIEL - Développement de logiciels avec UML 2 et OCL - Cours, études de cas et exercices corrigés, volume 6. Collection Technosup, edition ellipses edition, 2013.

[6] M. Bagherzadeh, K. Jahed, B. Combemale, and J. Dingel. Live-umlrt: A tool for live modeling of uml-rt models. In 2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C), pages 743–747, Sep. 2019.

[7] Franck Barbier and Eric Cariou. Executable modeling for reactive programming. In Slimane Hammoudi, Luís Ferreira Pires, and Bran Selic, editors, Model-Driven Engineering and Software Development - 6th International Conference, MODELSWARD 2018, Funchal, Madeira, Portugal, January 22-24, 2018, Revised Selected Papers, volume 991 of CCIS, pages 1–8. Springer, 2018.

[8] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. Model-Driven Software Engineering in Practice: Second Edition. Morgan & Claypool Publishers, 2nd edition, 2017.

[9] Federico Ciccozzi. On the automated translational execution of the action language for foundational uml. Software & Systems Modeling, 17(4):1311–1337, Oct 2018.

[10] Federico Ciccozzi, Ivano Malavolta, and Bran Selic. Execution of uml models: a systematic review of research and practice. Software & Systems Modeling, 18(3):2313–2360, Jun 2019.

[11] Sanford Friedenthal, Alan Moore, and Rick Steiner. A Practical Guide to SysML: Systems Modeling Language. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.

[12] Alexandre Gicquel, Antoine Guerin, and Rozen Anthony. Refinement of communication protocols by model transformation. Technical report, Master ALMA, University of Nantes, May 2021.

[13] Sahar Guermazi, Jérémie Tatibouet, Arnaud Cuccuru, Ed Seidewitz, Saadia Dhouib, and Sébastien Gérard. Executable modeling with fuml and alf in papyrus: Tooling and experiments. In Proc. of the 1st International Workshop on Executable Modeling in (MODELS 2015)., pages 3–8, Ottawa, Canada, September 2015.

[14] OMG. UML 2.3 Documentation, 2010.

[15] Simon Pickin, Claude Jard, Thierry Jéron, Jean-Marc Jézéquel, and Yves Le Traon. Test Synthesis from UML Models of Distributed Software. IEEE Transactions on Software Engineering, 33(4):252–268, 2007.

[16] Elena Planas, Jordi Cabot, and Cristina Gómez. Lightweight and static verification of uml executable models. Comput. Lang. Syst. Struct., 46(C):66–90, November 2016.

[17] P. Roques and F. Vallée. UML 2 en action: De l'analyse des besoins à la conception. Architecte logiciel. Eyrolles, 2011.

[18] Dionisie Rosca and Luísa Domingues. A systematic comparison of roundtrip software engineering approaches applied to uml class diagram. Procedia Computer Science, 181:861–868, 2021. International Conference on Health and Social Care Information Systems and Technologies 2020, CENTERIS/ProjMAN/HCist 2020.

[19] Gabriel Sebastián, Jose A. Gallud, and Ricardo Tesoriero. Code generation using model driven architecture: A systematic mapping study. Journal of Computer Languages, 56:100935, 2020.

[20] Ansgar Radermacher Van Cam Pham, Sébastien Gérard, and Shuai Li. Complete code generation from uml state machine. In Proceedings of the 5th International Conference on Model-Driven Engineering and Software Development, volume 1, pages 208–219, 2017.