Une méthode pour le développement collaboratif de systèmes embarqués

Nicolas Hili

UNIVERSITÉ DE GRENOBLE

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE GRENOBLE

Spécialité : **Informatique**

Arrêté ministériel : 7 août 2006

Présentée par

Nicolas Hili

Thèse dirigée par **Sophie Dupuy-Chessa** et **Dominique Rieu**
et co-encadrée par **Christian Fabre**

préparée au sein du **Laboratoire d'informatique de Grenoble**,
du **Commissariat à l'énergie atomique et aux énergies alternatives**
et de l'**École doctorale mathématiques, sciences et technologies de l'information, informatique**

Une méthode pour le développement collaboratif de systèmes embarqués

Thèse soutenue publiquement le **11 décembre 2014**,
devant le jury composé de :

M. Yves Ledru
Professeur à l'Université de Grenoble 1, Président
Mme Régine Laleau
Professeur à l'Université Paris-Est Créteil, Rapporteur
M. Mourad Chabane Oussalah
Professeur à la Faculté des Sciences de Nantes, Rapporteur
M. Hubert Dubois
Directeur de programme à CEA Tech Midi Pyrénées, Toulouse, Examinateur
M. Romain Guider
Ingénieur R&D à Obeo Nantes, Examinateur
Mme Sophie Dupuy-Chessa
Professeur à l'Université Pierre Mendès France, Grenoble, Directrice de thèse
Mme Dominique Rieu
Professeur à l'Université Pierre Mendès France, Grenoble, Co-Directrice de thèse
M. Christian Fabre
Ingénieur Chercheur au CEA LETI, Grenoble, Co-encadrant de thèse

Dedicação

Eu dedico minha tese a você, meu anjo.
Não tenho palavras para te agradecer e para te dizer como é grande o meu amor por você. Você nunca deixou de me encorajar e de me dar apoio. Você sempre esteve presente quando precisei e eu espero poder retribuir tudo o que você fez por mim.

Meu coração é seu.

Remerciements

Ce manuscrit est le fruit de trois années de thèse passées au sein du Laboratoire infrastructure et atelier logiciel pour puces (LIALP) du Département architecture conception et logiciels embarqués (DACLE) du CEA LETI et de l'équipe Systèmes d'Information - inGénierie et Modélisation Adaptables (SIGMA) du Laboratoire d'informatique de Grenoble (LIG). Aussi, je tiens tout d'abord à remercier Monsieur Vincent Olive et Madame Christine Verdier, respectivement directeur du LIALP et directrice de l'équipe SIGMA pour leurs accueils chaleureux au sein de leurs équipes. Je présente également mes profonds remerciements à Messieurs Thierry Collette et Marc Belleville, ainsi qu'à Madame Catherine Bour, respectivement chef, directeur scientifique et assistante RH du DACLE pour m'avoir accueilli chaleureusement et donné l'opportunité de travailler au sein du département.

Je tiens à témoigner toute ma gratitude à mes directrices de thèse, Mesdames Sophie Dupuy-Chessa et Dominique Rieu, ainsi qu'à mon encadrant de thèse Christian Fabre. Vous avez été exceptionnels avec moi et je ne vous remercierai jamais assez pour m'avoir soutenu durant ces trois années et pour m'avoir donné tous les moyens de réussir. Un grand merci pour vous être montrés présents au quotidien et pour tout ce que vous avez pu m'apprendre.

J'adresse mes sincères remerciements à Madame Régine Laleau et Monsieur Mourad Chabane Oussalah, respectivement professeurs à l'Université Paris-Est Créteil et à la Faculté des Sciences de Nantes, pour avoir accepté d'évaluer mon travail en qualité de rapporteurs. Je remercie également Monsieur Yves Ledru, professeur à l'Université de Grenoble 1, pour me faire l'honneur de présider ce jury de thèse, ainsi que Monsieur Hubert Dubois, directeur de programme à CEA Tech Midi Pyrénées et Monsieur Romain Guider, ingénieur R&D à Obeo Nantes pour leur participation en tant qu'examinateurs.

Ces trois années passées au LIALP et à l'équipe SIGMA ont été extrêmement enrichissantes tant sur le plan professionnel que sur le plan personnel et je tiens à remercier tous les membres et en particulier les permanents qui ont contribué à instaurer un environnement de travail admirable au sein des deux équipes.

Je tiens à remercier particulièrement Monsieur Julien Mottin et Monsieur Levent Gurgen, ingénieurs-chercheurs au CEA LETI pour m'avoir permis de participer à d'autres projets au sein de l'équipe LIALP.

Un grand merci à toi, Levent, pour m'avoir permis d'accomplir mon rêve de visiter le Japon. Ce séjour a été inoubliable pour moi et pour cela, je t'exprime toute ma gratitude, ainsi qu'à Messieurs Shinichi Honiden, Ishikawa Fuyuki et Kenji Tei pour leur chaleureux accueil au sein du laboratoire Honiden de National Institute of Informatics (NII) à Tokyo et sans qui ce séjour n'aurait été possible. Je remercie également Monsieur Marvin Ceschel avec qui j'ai eu le plaisir de travailler durant ce séjour.

Je tiens également à remercier Madame Annie Culet, maître de conférence à l'Université Pierre Mendès France, pour m'avoir offert l'opportunité de réaliser des enseignements à l'Institut universitaire de technologie (IUT) 2 à Grenoble.

Mes travaux de recherche m'ont amené à échanger avec différents doctorants, stagiaires et apprentis avec qui j'ai pris énormément de plaisir à travailler. Aussi, je tiens à remercier

spécialement Ivan, Fayçal, Yassine et Salma du côté du Commissariat à l'énergie atomique et aux énergies alternatives (CEA), ainsi qu'Anthony du côté de l'équipe SIGMA.

Sur le plan personnel, ces trois années m'ont permis de rencontrer de nombreuses personnes et aujourd'hui amis. Un grand merci à toi Raquel. Tu as apporté beaucoup de joie et de bonheur dans ma vie et je ne te remercierai jamais assez pour tout le soutien que tu m'as procuré. Merci également mes meilleurs amis, Mouna et Mahmoud, à qui je souhaite énormément de bonheur dans la vie.

Ma route a croisé celle de Synergy, l'association du personnel et des doctorants du LIG et je tiens à remercier toutes les personnes que j'ai pu y rencontrer : Elmehdi, Thomas, Lauren, Cécile, David, Carole, ...

Merci à mes collègues de bureau, anciens et actuels, Victor, Lionel, Yeter et Ivan pour tous les bons moments passés avec vous. Je remercie également Ozan, Molka, Olesia, Mai, Safietou, Jander, Wijdene, Emmanuelle, Élisabeth, Fernando, Jérôme, Oussama, Tiana, Isabelle, Amira, Juan Pablo, Mario, Aurélien, Pierre et tous ceux que je n'ai pas cités mais à qui je pense également très fort.

Enfin, mes derniers remerciements vont à toute ma famille. Je vous témoigne toute ma gratitude pour tout le soutien que vous avez pu m'apporter durant ces trois années.

Table des matières

Liste des figures

Chapitre 1

Introduction

Liste des sections

1.1 Contexte

En moyenne 230 systèmes embarqués sont utilisés quotidiennement par une personne [Sifakis 2011]. Les systèmes embarqués peuvent se définir comme des « *artéfacts d'ingénierie nécessitant du calcul et étant sujets à des contraintes physiques* » [Henzinger & Sifakis 2007]. Ils occupent une place de plus en plus importante dans notre vie quotidienne. Ils sont intégrés dans des ensembles plus vastes (cf. fig. 1) : borne de retrait de billets de train, calculateurs embarqués dans un avion, correcteur électronique de trajectoire dans une voiture, etc.

Figure 1 – Divers systèmes intégrant des systèmes embarqués

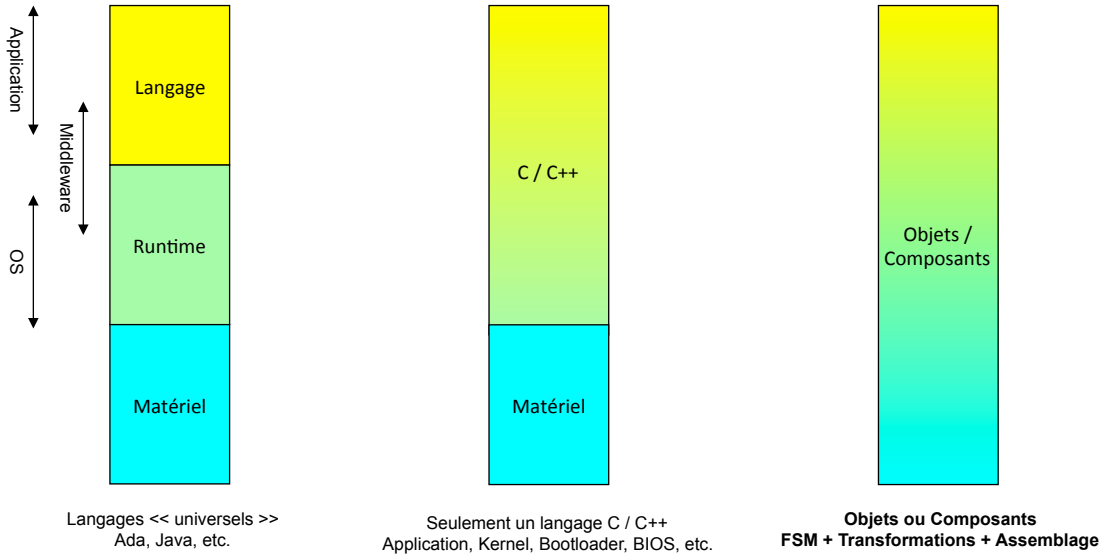


FIGURE 2 – Évolution du développement des systèmes embarqués

Comme le montre la figure 1, les systèmes embarqués s'intègrent dans un environnement physique dont il faut tenir compte pour leur développement. D'une part, l'interaction avec un processus physique externe impose des contraintes au niveau système en termes de performance, de précision et de prédictibilité. D'autre part, les calculateurs hébergeant le logiciel sont soumis à des contraintes industrielles. Ces contraintes portent sur différents aspects des systèmes tels que la sûreté, la minimisation des ressources matérielles pour des raisons de consommation énergétique, l'encombrement ou encore les coûts de développement et de fabrication. Ces contraintes imposent des architectures de calcul spécifiques et généralement non homogènes qu'il est nécessaire de prendre en compte lors du développement du système.

Dans ce contexte, plusieurs approches prenant en compte tout ou partie de ces contraintes ont permis d'aborder les systèmes embarqués au niveau de leur développement. La figure 2 illustre cette évolution. Les premiers développements s'appuyèrent sur une séparation stricte du système en plusieurs couches logicielles et matérielles, chaque couche possédant son langage dédié. Cette séparation très nette tirait ses origines des premiers co-développements logiciel et matériel [Gupta & De Micheli 1993, Ernst *et al.* 1993]. Elle permettait de partitionner les tâches effectuées en logiciel et en matériel pour en tirer un maximum de bénéfice (paramétrisation, consommation énergétique, etc.), mais introduisait un manque de flexibilité au niveau du développement, dû aux problèmes de compatibilité entre les différentes couches. Un effort d'unification des langages a été réalisé pour le logiciel, permettant plus de flexibilité tout en conservant un maximum d'indépendance à la plate-forme d'exécution (approche du milieu de la figure 2). Cette approche marque toujours une discontinuité entre le développement logiciel et le développement matériel et nécessite le développement d'une interface entre le logiciel et le matériel. La dernière approche traduit un effort d'homogénéisation du développement logiciel et matériel. Cet effort a été rendu possible avec l'usage de modèles à composants et de l'ingénierie dirigée par les modèles. Il supprime les discontinuités entre les couches, augmente grandement la flexibilité du développement et offre des possibilités d'optimisation globales du système.

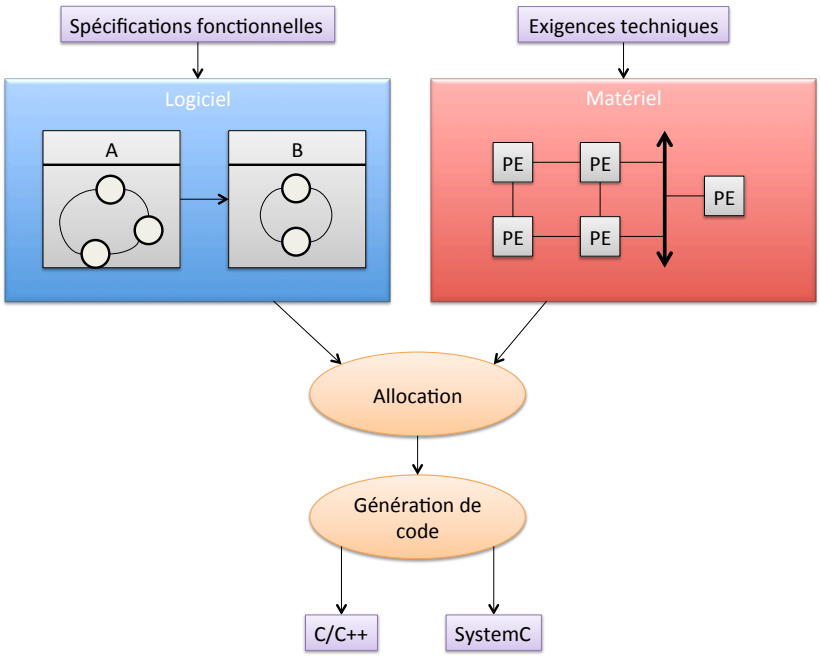


FIGURE 3 – Le développement des systèmes embarqués

Les approches basées sur les modèles permettent un développement homogène et des analyses qualitatives et quantitatives durant les phases de conception, améliorant la qualité du système conçu. Les langages de modélisation basés sur UML [1] ou AADL [2] ont ainsi permis d'obtenir une indépendance entre la sémantique d'exécution et le langage de programmation choisi [Henzinger & Sifakis 2007]. Ces langages de modélisation sont généralement intégrés dans des démarches proposant un développement de l'application et une allocation des différents fragments de l'application sur les composants matériels d'une plate-forme (cf. fig. 3). L'application suit une approche descendante *par raffinement* en partant d'une formalisation des besoins vers une conception indépendante de la partie physique. Elle est généralement modélisée à l'aide de modèles structurels (objets / composants) et dynamiques (e.g. machines à états). La partie physique est quant à elle souvent définie comme l'assemblage de composants définis en langage de description matérielle (e.g. IP-XACT) et répondant à différentes exigences techniques, telles que le débit maximum disponible, la vitesse des processeurs, les temps de latence. Une phase d'allocation permet d'allouer les objets applicatifs aux différents composants de la partie physique. Enfin, une dernière phase consiste à générer le code de l'application (C/C++, SystemC) sur des architectures matérielles à l'aide de générateurs de code spécifiques.

Ces approches mettent principalement l'accent sur le développement logiciel et présupposent de connaitre l'architecture matérielle destinée à héberger le logiciel. D'autres approches consistent à considérer la *plate-forme* qui accueille la partie applicative du système embarqué. La plate-forme peut se définir comme « *une librairie de composants pouvant être assemblés pour générer une conception à un niveau d'abstraction donné* » [Sangiovanni-Vincentelli 2007]. La

1. En anglais, Unified Modeling Language (UML).
2. En anglais, Architecture Analysis and Design Language (AADL).

plate-forme n'est plus seulement la partie matérielle du système embarqué mais peut aussi désigner une plate-forme logicielle (e.g. système d'exploitation temps-réel). Les plates-formes sont conçues en couches dans une approche ascendante afin de « rencontrer » l'application conçue dans une approche descendante. Les approches centrées sur l'usage intensif des plates-formes permettent de favoriser la réutilisation, d'assurer la qualité du système et de réduire les coûts de production. Elles sont cependant souvent limitées dans la définition de la plate-forme, réduite à un simple assemblage de composants, ne permettant pas d'étudier l'impact de ces assemblages sur l'hébergement et l'exécution d'applications constituant les systèmes embarqués.

1.2 Problématique

Les différentes contraintes de développement des systèmes embarqués dues aux spécificités de ces systèmes et du contexte industriel auquel ils prennent part complexifient d'autant leur développement et nécessitent de formaliser les phases de développement prenant en compte leur différentes spécificités (co-développement logiciel et matériel, exécution sur une plate-forme, aspect temps-réel, certification des développements, etc.) des systèmes embarqués.

Les notions de méthode et de processus de développement qui abordent le problème de description des activités à réaliser ne sont cependant pas bien connues dans le domaine de l'ingénierie des systèmes embarqués. Elles sont en revanche bien maitrisées dans le domaine de l'ingénierie des systèmes d'information. Les recherches et pratiques dans ce domaine ont su établir les fondements d'une véritable ingénierie des méthodes permettant de concevoir, formaliser, outiller, aussi bien les langages de modélisation que les processus de développement. Nous avons donc cherché à identifier comment l'expérience acquise dans ce domaine peut servir à définir une méthode formalisée et outillée prenant en compte les contraintes des systèmes embarqués. La problématique que nous nous posons donc dans le cadre de cette thèse est la suivante :

> « *Comment définir, formaliser et outiller une méthode de développement de systèmes embarqués permettant de réduire la complexité de leur développement en prenant en compte un certain nombre de leur spécificités, telles que les interactions physiques, l'exécution sur une plate-forme ou la mesure et le pilotage du développement, par l'application des techniques de l'ingénierie des systèmes d'information ?* »

1.3 Objectifs et contributions

Les travaux présentés dans ce mémoire ont pour objectif d'exploiter et adapter les acquis de l'ingénierie dirigée par les méthodes dans le contexte de la conception des systèmes embarqués. Ces travaux ont permis la formalisation d'une méthode nommée $\langle \mathsf{HOE} \rangle^2$, signifiant « *Highly Heterogeneous Object-Oriented Efficient Engineering* ». Nous avons orienté ces travaux autour de quatre principaux objectifs :

Objectif 1. Le premier objectif de la thèse a été de définir et formaliser le processus $\langle \mathsf{HOE} \rangle^2$ en prenant en compte certaines spécificités des systèmes embarqués (interactions physiques, exécution sur une architecture limitée en nombre de ressources, etc.) et de leur ingénierie (développement collaboratif, mesure et pilotage du développement, etc.). La définition de la méthode

$\langle$HOE$\rangle^2$ a impliqué la formalisation de son processus, en explicitant les différentes activités et rôles des participants, ainsi que les différents points de synchronisation permettant la mise en commun des développements de l'application et de l'architecture.

Objectif 2. Le second objectif de la thèse a été d'intégrer la notion de *plate-forme* hébergeant la partie applicative du système embarqué et de proposer une extension du processus afin de prendre en compte son développement. Cette extension permet d'une part le développement de la plate-forme en suivant le même flot d'activités que l'application, et d'autre part le découpage du flot de développement de plates-formes complexes en plusieurs flots avec des points de synchronisation entre les différents flots. Cet objectif a nécessité d'expliciter comment une plate-forme complexe peut être considérée comme une composition de plates-formes plus simples et quel en est l'impact sur le code applicatif généré.

Objectif 3. La gestion de projet et la traçabilité intégrées au sein du processus a constitué le troisième objectif de la thèse. Cet objectif aborde différentes spécificités de l'ingénierie des systèmes embarqués telles que le besoin de mesurer et de piloter l'avancement de développement ou le besoin de traçabilité au sein du processus pour des questions de certification. Nous avons pour cela introduit des concepts relatifs à la gestion de projet et à la planification des activités dans le langage, et nous avons établi le lien entre les activités de l'ingénierie des systèmes embarqués et les modèles produits.

Objectif 4. Afin d'outiller la méthode prenant en compte les spécificités de l'ingénierie des systèmes embarqués telles que le travail collaboratif, le dernier objectif a couvert le développement d'un outil multi-utilisateurs et dédié à la méthode $\langle$HOE$\rangle^2$. Cet outil permet les supports du langage et du processus, tout en assurant la synchronisation des développements et l'intégration de la gestion de projet permettant au chef de projet de planifier les développements et d'organiser ses équipes.

1.4 Plan du mémoire

Le mémoire a été organisé en deux parties. La première partie est constituée d'**états de l'art** sur l'ingénierie dirigée par les méthodes et les systèmes embarqués. Cette partie a été organisée en trois chapitres :

Chapitre 2 (L'ingénierie des systèmes embarqués). Ce chapitre aborde l'étude de l'état de l'art des systèmes embarqués. Nous distinguons dans ce chapitre les différentes spécificités des systèmes embarqués et de leur ingénierie, spécificités qu'il est nécessaire de prendre en compte dans une méthode.

Chapitre 3 (L'ingénierie des méthodes et des modèles). Ce chapitre aborde l'étude de l'état de l'art de l'ingénierie des méthodes et des modèles. Dans ce chapitre, nous posons les définitions d'une méthode de développement formalisée et outillée, d'un modèle de processus et nous étudions les techniques de modélisation de processus.

Chapitre 4 (Les méthodes de développement de systèmes embarqués). Ce chapitre établit le pont entre les états de l'art de l'ingénierie des méthodes et des systèmes embarqués. Nous identifions ainsi les caractéristiques des processus, langages et outils des méthodes nécessaires pour prendre en compte les spécificités des systèmes embarqués et de leur ingénierie. Nous étudions ensuite quelques méthodes de développement vis-à-vis de ces caractéristiques afin d'identifier les lacunes de ces dernières et établir nos différents points de contribution.

La seconde partie de ce mémoire présente **notre contribution** prenant en compte les différents objectifs précédemment présentés. Cette partie est structurée en six chapitres :

Chapitre 5 (Présentation générale de la méthode $\langle$HOE$\rangle^2$). Ce chapitre introduit brièvement la méthode $\langle$HOE$\rangle^2$ et en délimite les différentes spécificités pour aborder la problématique identifiée.

Chapitre 6 (Description détaillée de la méthode $\langle$HOE$\rangle^2$). Ce chapitre détaille la formalisation du processus de la méthode et du langage de modélisation. Il présente une version restreinte du processus ne prenant pas en compte la composition des plates-formes.

Chapitre 7 (Systèmes embarqués et composition de plates-formes). Ce chapitre présente une extension du processus et du langage afin de prendre en compte la composition de plates-formes au sein de la méthode $\langle$HOE$\rangle^2$.

Chapitre 8 (Gestion de projet et traçabilité couplées dans le processus). Ce chapitre aborde l'intégration de la traçabilité et de la gestion de projet. La contribution présentée permet d'une part d'assurer la traçabilité entre les modèles conçus et les activités de modélisation, et d'autre part d'offrir au chef de projet des outils efficaces pour mesurer et piloter l'avancement du développement et organiser ses équipes de développement.

Chapitre 9 (CanHOE2, un atelier de développement dédié). Ce chapitre présente les développements réalisés pour l'outillage de la méthode $\langle$HOE$\rangle^2$. L'outil réalisé se nomme CanHOE2 (pour *CANonical* $\langle$*HOE*$\rangle^2$). Il est dédié et permet le support du langage ainsi que du processus. Il offre un environnement de développement multi-utilisateurs et intègre des vues de gestion de projet à l'attention du chef de projet.

Chapitre 10 (Validation). Ce chapitre présente une étude de cas permettant de vérifier certaines propriétés de la méthode. Dans ce chapitre, nous présentons d'une part comment la méthode $\langle$HOE$\rangle^2$ aborde la transformation des modèles de l'application indépendants de la plate-forme en des modèles spécifiques à cette dernière, et d'autre part comment, à partir de ces modèles spécifiques, nous pouvons générer du code pour ces plates-formes cibles à l'aide de générateurs de code spécifiques.

Chapitre 11 (Conclusion et Perspectives). Ce dernier chapitre résume les contributions réalisées et introduit les différentes perspectives de recherche.

Première partie

État de l'art

Liste des chapitres

Introduction à l'état de l'art

Cette partie vise à présenter l'état de l'art des systèmes embarqués ainsi que celui de leur ingénierie conjointement à celui des méthodes de développement. Elle est divisée en trois chapitres :

Chapitre 2 (L'ingénierie des systèmes embarqués). Ce chapitre présente l'ingénierie des systèmes embarqués. Il introduit la définition d'un système embarqué afin de pouvoir en déterminer les spécificités fondamentales. Leur étude, ainsi que celle du contexte industriel nous permettra d'identifier les enjeux de l'ingénierie des systèmes embarqués.

Chapitre 3 (L'ingénierie des méthodes et des modèles). Ce chapitre présente l'état de l'art de l'ingénierie des méthodes et des modèles. Il pose les définitions d'une méthode de développement, d'un modèle de processus et présente les requis nécessaires afin de définir une méthode de développement formalisée et outillée.

Chapitre 4 (Les méthodes de développement de systèmes embarqués). Ce chapitre permet de faire le lien entre les deux chapitres précédents. Il présente les caractéristiques des méthodes de développement des systèmes embarqués. Il établit une étude comparative des méthodes de développement actuelles afin de voir comment sont réellement prises en compte les spécificités des systèmes embarqués. Il se termine par une synthèse des forces et faiblesses des méthodes actuelles et notre positionnement.

Chapitre 2

L'ingénierie des systèmes embarqués

Liste des sections

Nous pouvons trouver dans la littérature plusieurs définitions d'un système embarqué. Les définitions les plus pertinentes sont celles de Joseph Sifakis et Thomas A. Henzinger [Henzinger & Sifakis 2007] et de Alberto Sangiovanni-Vincentelli [Sangiovanni-Vincentelli 2007]. Pour Henzinger et Sifakis, un système embarqué est défini de la façon suivante :

> « Un système embarqué est un artéfact d'ingénierie nécessitant du calcul et étant sujet à des contraintes physiques [...] Les contraintes physiques interviennent aux travers de deux types d'interaction entre le processus informatique et le monde physique : réaction à un environnement physique et exécution sur une plate-forme physique. [Henzinger & Sifakis 2007] »

Cette définition fait intervenir les notions d'*environnement* et de *contraintes physiques* que l'on retrouve également dans la définition de Sangiovanni-Vincentelli :

> « Un système embarqué est un système dédié dans lequel l'unité de calcul est complètement contenue dans un système qu'elle contrôle [...]. D'un point de vue technique, un système embarqué interagit avec l'environnement qui l'entoure d'une manière contrôlée satisfaisant un ensemble de contraintes de réactivité en termes de qualité et de rapidité. [Sangiovanni-Vincentelli 2007] »

1 Les spécificités des systèmes embarqués

Afin de pouvoir étudier les caractéristiques que doivent posséder les méthodes de développement des systèmes embarqués, il est nécessaire de pouvoir définir les spécificités de ces derniers. Nous étudions dans cette section les spécificités des systèmes embarqués que nous retrouvons dans la littérature et que nous présentons suivant six perspectives dans la figure 4.

Instrumentation du monde physique. Un système embarqué est en interaction permanente avec le monde physique. Deux aspects de cette interaction sont étudiés par de nombreux auteurs [Kopetz 1997, Lee 2005, Broy 2006a, Henzinger & Sifakis 2007, Sifakis 2011]. Le premier est lié au fait que le système embarqué est en situation de contrôle / commande avec le monde physique et qu'il doit le faire à un rythme, une précision et une fiabilité compatibles avec le processus physique ainsi contrôlé / instrumenté. Le second aspect correspond à l'exécution du

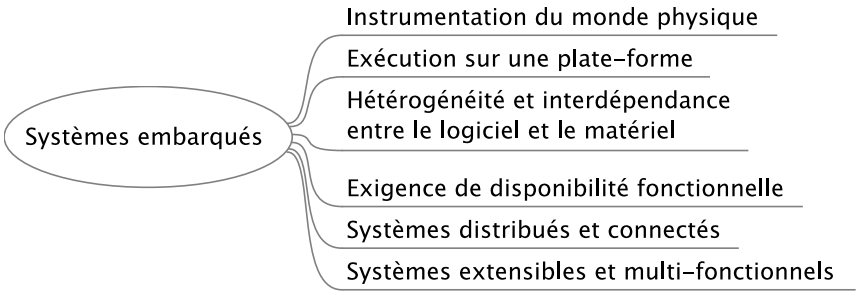


FIGURE 4 – Les spécificités des systèmes embarqués

code sur une plate-forme physique ayant certaines capacités que nous recherchons toujours à réduire. L'instrumentation du monde physique relève du fonctionnel d'un système embarqué tandis que l'exécution sur une plate-forme relève des techniques de réalisation. Ces deux aspects sont orthogonaux et seront traités comme deux perspectives distinctes.

Dans [Kopetz 1997], les interactions physiques réfèrent aux besoins fonctionnels des systèmes temps-réel. Les besoins fonctionnels sont divisés en la collecte de données temps-réel, le contrôle digital direct et l'interaction homme-machine – cette dernière notion est reprise dans [Broy 2006a]. L'observation de ces données, ou entités temps-réel, permet de définir l'état d'un système à un instant donné. Ces données sont observables mais non modifiables, collectées à partir de capteurs réels. Le contrôle digital concerne les entités calculées qui alimentent les actionneurs dans le but de contrôler le système temps-réel. Afin que les données observées soient observables, les données contrôlées contrôlables, la dimension humaine est prise en compte. D'une part, le système doit informer l'utilisateur de son état courant à partir des capteurs et d'autre part l'utilisateur peut contrôler cet état au moyen d'actionneurs. Les besoins fonctionnels réfèrent donc au besoin d'interagir avec le monde physique, par son observation et son contrôle. Lee [Lee 2005] discute de la non-trivialité de l'interaction entre le logiciel embarqué et le monde physique – réaction du logiciel aux données recueillies par les capteurs et envoi des commandes aux actionneurs. Henzinger et Sifakis [Henzinger & Sifakis 2007, Sifakis 2011] définissent l'*environnement physique* avec lequel le système *réagit*, réaction toutefois contrainte par des grandeurs physiques (temps de réponse, latence, etc.).

La prise en compte de l'interaction avec le monde physique introduit des contraintes complexifiant le développement des systèmes embarqués et supprimant la barrière entre le logiciel et le monde physique [Henzinger & Sifakis 2007]. Nous garderons donc comme perspective l'*instrumentation du monde physique* qui relève de la mesure et du contrôle – le plus souvent temps-réel – de l'environnement physique aux travers de capteurs et d'actionneurs.

Exécution sur une plate-forme. La partie physique d'un système embarqué comprend la *plate-forme* physique et les ressources que cette dernière met à disposition pour l'exécution d'un système embarqué. Henzinger et Sifakis [Henzinger & Sifakis 2007, Sifakis 2011] définissent la *plate-forme* sur laquelle un système embarqué s'*exécute*. L'*exécution* sur une plate-forme introduit des contraintes physiques liées à la vitesse des processeurs, les débits de communication dans les bus de données et est sujette à des possibles dysfonctionnements matériels.

Les ressources physiques d'un système embarqué sont un point crucial influençant tous les autres critères d'un système embarqué : dimension, poids, consommation, fiabilité, réactivité,

coût de fabrication, etc. Une augmentation de ressources offre une réactivité et une fiabilité accrues au système, mais augmente également sa consommation et son coût de fabrication tout en réduisant son autonomie pour les systèmes sur batterie. Inversement, une diminution de ressources engendre une réduction des coûts de fabrication et une consommation diminuée au détriment de la qualité de service du système. Les systèmes multi-fonctionnels [Broy 2006a] nécessitent que les ressources soient suffisamment surdimensionnées pour supporter un large nombre d'applications, tandis que les systèmes sur batterie nécessitent qu'elles soient suffisamment minimisées pour satisfaire les contraintes en terme de consommation. La dimensionnement des ressources physiques traduit donc un compromis devant être réalisé pour toutes les classes de systèmes, des systèmes critiques à faible complexité aux systèmes non-critiques, à forte complexité et souvent extensibles et distribués [Sifakis 2011]. Notamment, il est d'usage de surdimensionner les systèmes critiques pour augmenter leur fiabilité et de réduire les ressources d'un système complexe et distribué à grande échelle afin d'en minimiser le coût tout en conservant une disponibilité des services acceptable [Sifakis 2011]. Pour certains systèmes, ce compromis à réaliser est très important et est très souvent dicté par le coût de fabrication et l'économie de l'échelle. Par exemple, il prend tout son sens pour les smartphones dont les ventes ont atteint 435 millions d'unités vendues en 2013 [Gartner 2013]. La maitrise des coûts de fabrication est directement induite par la minimisation des ressources et de la partie physique. En plus des coûts de fabrication, les coûts de fonctionnement influencent également le compromis à réaliser sur la quantité de ressources à embarquer au sein d'un système embarqué. Dans le cas d'un avion par exemple, toute économie réalisée en termes de dimension, poids, place peut se traduire par une augmentation de l'autonomie en vol, une diminution de la consommation de carburant ou bien encore l'augmentation du nombre de sièges disponibles à bord de l'avion.

Les différentes considérations effectuées dans cette partie nous amène à considérer la plateforme, ses ressources et leur minimisation au regard de l'exécution d'un système comme étant les points fondamentaux reliant ces différents besoins. Nous les regroupons sous la perspectives de l'*exécution sur une plate-forme*.

Hétérogénéité et interdépendance entre le logiciel et le matériel. L'hétérogénéité est discutée dans [Wolf 2003, Broy 2006a, Sangiovanni-Vincentelli 2007, Henzinger & Sifakis 2007, Wolf *et al.* 2008, Sifakis 2011]. Elle revêt différentes formes. L'hétérogénéité de *distribution* pour les systèmes embarqués distribués est discutée dans [Broy 2006a]. Elle relève de la capacité du système embarqué à se connecter à d'autres systèmes, au moyen d'Internet par exemple. Sifakis [Sifakis 2011] définit quant à lui l'hétérogénéité du *calcul* (synchrone ou asynchrone), *d'interaction* (sémaphore, rendez-vous, broadcast, etc.). Dans [Henzinger & Sifakis 2007], les auteurs parlent également d'hétérogénéité *temporelle* (temps discret ou continu). Néanmoins, le terme *hétérogénéité* renvoie le plus souvent à *l'interdépendance entre le logiciel et le matériel*. Elle tire son origine des premiers co-développements apparus dans les années 90 avec notamment les travaux de Gupta [Gupta & De Micheli 1993] et de Ernst [Ernst *et al.* 1993]. L'interdépendance entre le logiciel et le matériel complexifie le développement des systèmes embarqués [Broy 2006a]. Henzinger et Sifakis [Henzinger & Sifakis 2007] définissent comme principale difficulté la *composition* de composants hétérogènes.

Parmi toutes les dimensions appliquées à l'hétérogénéité, *l'hétérogénéité et l'interdépendance entre le logiciel et le matérielle* est selon nous une spécificité à part entière très largement traitée. Nous la retiendrons comme étant à elle seule une perspective majeure à étudier.

Exigence de disponibilité fonctionnelle. La *disponibilité fonctionnelle* est une exigence importante pour les systèmes embarqués. Plusieurs critères permettent de la qualifier : réactivité, fiabilité, criticité, et autonomie. D'un point de vue de l'autonomie, un système embarqué doit pouvoir assurer un service continu sans intervention humaine [Sifakis 2011, Ramesh *et al.* 2012]. Dans [Sifakis 2011], Sifakis associe également la disponibilité fonctionnelle à la réactivité, qu'il définit comme étant la capacité à répondre en un temps déterminé et borné. Cette dernière est affectée par les temps des processus externes des actionneurs et des capteurs et donc par l'environnement physique entourant le système embarqué [Broy 2006a]. La *fiabilité* est décrite par Kopetz [Kopetz 1997], Lee[Lee 2005], Broy [Broy 2006a] et Sifakis [Sifakis 2011]. Selon Sifakis, la fiabilité est définie comme l'invulnérabilité des systèmes embarqués en cas d'attaques ou de pannes matérielles. [Kopetz 1997] réfère à la probabilité que possède un système embarqué d'assurer un service durant un temps déterminé sans dommage ou panne. Selon Kopetz, elle couvre plusieurs dimensions telles que la sureté, la sécurité, la maintenabilité et la disponibilité. Selon Broy, la fiabilité est liée à la criticité [Broy 2006a]. Il partage le même avis que Lee sur l'importance de la fiabilité dans les systèmes embarqués et notamment dans les systèmes critiques, où la non-réponse peut mettre en cause la sécurité et avoir des conséquences dramatiques [Lee 2005, Broy 2006a]. Ces quatre critères – la réactivité, la fiabilité, la criticité et l'autonomie – permettent de qualifier comme une seule et même perspectives les *exigences de disponibilité fonctionnelles* que nous étudierons par la suite.

Systèmes distribués et connectés. Les systèmes embarqués sont souvent perçus comme des *systèmes distribués inter-connectés* [Tanenbaum & Steen 2006, Broy 2006a], devant partager des ressources communes et interagir dans des délais très courts. Une voiture moderne possède plus de cent systèmes embarqués, couvrant la motorisation, la transmission, l'assistance à la conduite, le confort et la sécurité des passagers [Broy 2006b]. Tanenbaum et Steen [Tanenbaum & Steen 2006] définissent un *système distribué* comme étant une *collection indépendante d'ordinateurs apparaissant pour ses utilisateurs sous la forme d'un seul système cohérent.* Cette définition met en valeur deux concepts, celui du *composant* et du *système.* Dans cette définition le *composant* est un ordinateur, mais les auteurs soulignent que ce concept peut référer aussi bien à un ordinateur personnel qu'à un nœud dans un réseau de capteurs. Ainsi, un système embarqué peut être considéré comme un composant et une collection de systèmes embarqués comme le système distribué cohérent. La distribution complexifie le développement et le fonctionnement des systèmes embarqués, devant d'une part être vus dans leur indépendance et d'autre part formant un ensemble cohérent. Ils impliquent un certain nombre de contraintes et de problématiques, telles que le partage et la mise à disposition des ressources, la transparence de distribution et d'accès. Ils doivent être compatibles pour pouvoir communiquer et partager des modèles de données complexes, très souvent dans un contexte temps-réel, nécessitant des temps d'accès et de réponse très rapides, contraignant fortement le développement de l'application (code non bloquant, système de priorité, sémaphores, etc.). Ils doivent conserver un maximum d'indépendance afin d'anticiper et de prévenir des dysfonctionnements majeurs en cas de panne ou de non réponse de l'un des constituants du système. Réactivité, interconnection, communication, assurance de la qualité, intégration du système, sont autant de caractéristiques très largement contraintes par la distribution. Regarder le système embarqué comme un ensemble cohérent complexifie son développement et l'inter-connection de ses fonctionnalités. Ainsi, nous considérerons par la suite les *systèmes distribués et connectés* comme une spécificité majeure à prendre en compte pour le développement des systèmes embarqués.

Systèmes extensibles et multi-fonctionnels. L'*extensibilité* est traitée dans [Tanenbaum & Steen 2006, Sifakis 2011]. Sifakis définit l'*extensibilité d'un système* comme étant la capacité de gagner en fonctionnalité au prix éventuel d'une augmentation de ressources [Sifakis 2011]. L'extensibilité impose des contraintes sur les ressources d'un système embarqué [Ramesh *et al.* 2012]. La *multi-fonctionnalité* est une spécificité identifiée par Broy [Broy 2006a]. L'arrivée du logiciel embarqué a permis de développer des produits multi-usages, comme nous le voyons par exemple avec les téléphones mobiles. En effet, ces derniers, dont l'usage primaire était de téléphoner, deviennent de véritables ordinateurs permettant de prendre des photos, regarder des films ou bien se connecter à Internet. Prendre en compte la possibilité d'embarquer plusieurs fonctionnalités ajoute une nouvelle dimension dans le développement de ces systèmes.

L'*extensibilité* est une caractéristique particulièrement importante pour les systèmes *multi-fonctionnels* [Broy 2006a]. Ces systèmes nécessitent de surdimensionner la quantité de ressources afin de permettre l'ajout de nouvelles fonctionnalités [Sifakis 2011]. A contrario, les systèmes offrant un nombre limité et fixe de fonctionnalités nécessitent d'être dimensionnés selon leur besoin propre en ressources. Cette dernière classe de systèmes est liée au niveau de fiabilité attendu du système. En effet, un système critique à faible nombre de fonctionnalités qui ne saurait souffrir d'aucune panne peut être surdimensionné (redondance des ressources par exemple) afin d'améliorer sa fiabilité et sa disponibilité [Sifakis 2011]. Nous pensons qu'il est important d'associer l'*extensibilité* et la *multi-fonctionnalité* des systèmes, ces deux spécificités étant fortement interdépendantes. En effet, l'extensibilité traduit un compromis qui doit être réalisé entre la puissance de calcul nécessaire, le nombre d'applications supportées, les exigences de dimensionnement, de poids et d'autonomie. Pour un téléphone portable, une tablette tactile, un appareil photo, ces exigences sont en effet fondamentales. Ce compromis est moins important pour les systèmes critiques où la fiabilité du système prime sur toutes les autres spécificités, par exemple, le contrôle moteur d'une voiture. Ainsi, nous retiendrons comme perspective à étudier les *systèmes extensibles et multi-fonctionnels.*

Synthèse des points de vue

Cette partie synthétise les points de vue des différents auteurs cités dans la littérature autour des six axes que nous avons définis. Dans la table 1, nous avons résumé ces différents points de vue. Chaque ligne correspond à un travail cité. Chaque colonne correspond à une perspective étudiée. Dans chacun des champs, un ou plusieurs mots-clés résument le point de vue d'un auteur sur la perspective citée. Ces axes qualifient la *complexité* des systèmes embarqués. Tous les auteurs s'accordent pour dire qu'elle ne cesse de croître [Kopetz 1997, Broy 2006a, Sangiovanni-Vincentelli 2007, Henzinger & Sifakis 2007, Gamatié *et al.* 2011, Sifakis 2011]. Sangiovanni-Vincentelli et Broy expliquent son origine par la complexité croissante du logiciel embarqué, de l'architecture matérielle et de l'intégration des deux [Broy 2006a, Sangiovanni-Vincentelli 2007]. Gamatié et Kopetz expliquent que la complexité est engendrée par la diversité des systèmes embarqués [Kopetz 1997, Gamatié *et al.* 2011]. Sifakis et Henzinger estiment que cette complexité est majoritairement due au lien entre le système embarqué et l'environnement physique qu'il pilote et mesure, ainsi qu'à la fiabilité requise [Henzinger & Sifakis 2007].

Les spécificités des systèmes embarqués étudiées ne suffisent pas pour établir objectivement les caractéristiques que doivent posséder les processus et langages pour le développement de systèmes embarqués. Dans la section suivante, nous avons effectué le même travail d'analyse de l'état de l'art pour *l'ingénierie des systèmes embarqués.*

Travaux existants	Instrumentation du monde physique	Exécution sur une plate-forme	Hétérogénéité & Interdépendance entre le logiciel et le matériel	Exigence de disponibilité fonctionnelle	Systèmes distribués et connectés	Systèmes extensibles et muti-fonctionnels
	Perspectives étudiées					
[Kopetz 1997]	▸ **T**emps-réel ▸ **O**bservation ▸ **A**cquisition ▸ **C**ontrôle			▸ **F**iabilité		
[Lee 2005]	▸ **L**ogiciel embarqué ▸ **R**éaction ▸ **C**ommande			▸ **F**iabilité ▸ **S**ureté		
[Tanenbaum and Steen 2006]					▸ **C**ollection de composants ▸ **S**ystème cohérent	▸ **E**xtensibilité des systèmes distribués
[Broy 2006a]	▸ **I**nteraction ▸ **H**omme-Machine		▸ **L**ogiciel / matériel ▸ **D**istribution	▸ **C**apteurs / Actionneurs ▸ **F**iabilité / maintenabilité		▸ **P**lusieurs applications sur une seule et même puce ▸ **L**ogiciel embarqué
[Sangiovanni-Vincentelli 2007]		▸ **P**late-forme ▸ **I**ntégration	▸ **H**étérogénéité du matériel			
[Henzinger and Sifakis 2007]	▸ **M**esure ▸ **R**éaction	▸ **E**xécution ▸ **P**late-forme	▸ **L**ogiciel / matériel ▸ **S**ync / async. ▸ **D**iscret / continu	▸ **R**ésistance aux attaques et pannes ▸ **D**isponibilité en cas d'attaques		
[Sifakis 2011]	▸ **M**esure ▸ **R**éaction	▸ **E**xécution ▸ **P**late-forme ▸ **O**ptimisation des ressources	▸ **L**ogiciel / matériel ▸ **S**ync / async. ▸ **D**iscret / continu ▸ **A**bstraction	▸ **R**éactivité ▸ **S**ervice continu ▸ **S**ervice autonome		▸ **E**xtensibilité matérielle ▸ **A**ugmentation capacité / ressources

TABLE 1 – Points de vue des auteurs sur les spécificités des systèmes embarqués

2 Les spécificités de leur ingénierie

Les spécificités de l'ingénierie des systèmes embarqués ont été étudiées par [Gérard 2000, Sangiovanni-Vincentelli *et al.* 2004, Broy 2006b, Broy 2006a, Sangiovanni-Vincentelli 2007, Henzinger & Sifakis 2007, Biehl 2010, Biehl 2011, Sindico *et al.* 2012, Törngren 2013b]. Les travaux de Gérard et de Biehl sont focalisés sur le développement des systèmes embarqués dans le domaine automobile. Broy traite quant à lui les spécificités de l'ingénierie des systèmes embarqués d'une part dans le secteur automobile [Broy 2006b] et d'autre part dans un contexte plus général [Broy 2006a]. Il nous a paru intéressant de citer les travaux de Broy et Biehl à titre de comparaison, au sens où les idées des deux auteurs ciblent les mêmes spécificités mais selon des opinions divergentes. Nous résumons ces spécificités autour de six perspectives illustrées par la figure 5.

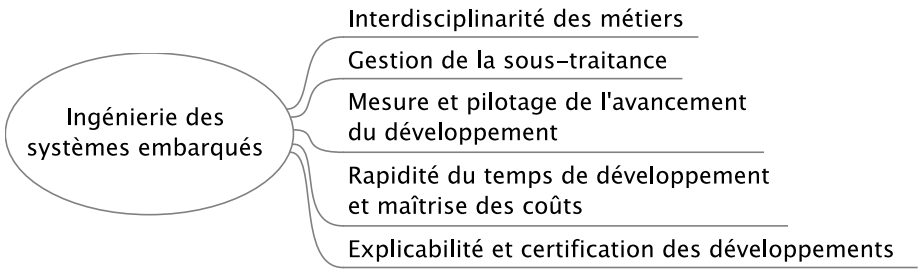


FIGURE 5 – Les spécificités de l'ingénierie des systèmes embarqués

Interdisciplinarité des métiers. L'ingénierie des systèmes embarqués est pluridisciplinaire selon [Gérard 2000, Broy 2006a, Biehl 2010, Henzinger & Sifakis 2007, Sifakis 2011]. Elle fait intervenir d'un côté des métiers communs à tous les secteurs d'ingénierie – tels que des experts du logiciel, du matériel, du contrôle – mais aussi des experts du domaine métier [Broy 2006a, Biehl 2010]. Ainsi par exemple, Gérard identifie certains rôles spécifiques au domaine de l'automobile : les constructeurs automobiles (parmi lesquels figurent les experts de l'automatique) et les fournisseurs (ou les équipementiers). Sangiovanni-Vincentelli identifie les mêmes rôles pour l'automobile, en distinguant toutefois à différentes échelles les fournisseurs de sous-systèmes, de composants et les fabricants [Sangiovanni-Vincentelli 2007].

Les visions de Broy et de Biehl semblent différentes. Broy présente l'interdisciplinarité des équipes comme un requis pour le développement des systèmes embarqués [Broy 2006a]. Selon lui, ces experts doivent travailler ensemble, utilisant les mêmes modèles et les mêmes théories comme support de communication entre les différents métiers. Biehl différencie en s'appuyant sur les travaux de Augsburg [Augsburg 2005] l'interdisciplinarité et la multidisciplinarité des équipes de développement [Biehl 2010] : les différentes disciplines sont intégrées dans une ingénierie interdisciplinaire ou demeurent séparées dans une ingénierie multidisciplinaire. Ainsi, chaque discipline est indépendante des autres et possède ses propres outils dédiés. Biehl se différencie donc de Broy sur cette vision.

Les positions d'Henzinger et de Sifakis se rapprochent de celle de Broy, au sens où ils sont confiants sur la définition d'une fondation scientifique commune à l'informatique et l'électronique [Henzinger & Sifakis 2007]. Sangiovanni-Vincentelli partage également cette opinion dans [Sangiovanni-Vincentelli 2007].

Tous les auteurs cités s'accordent donc pour dire que l'ingénierie des systèmes embarqués est pluridisciplinaire. En reprenant la distinction entre multi et interdisciplinarité apportée par Biehl [Biehl 2010], nous pouvons dire que l'ingénierie des systèmes embarqués est selon Biehl multidisciplinaire alors que Broy, Henzinger, Sifakis et Sangiovannis-Vincentelli penchent vers l'interdisciplinarité de l'ingénierie. Si la question ne semble pas simple à résoudre, due au nombre de métiers et de profils différents impliqués dans le développement des systèmes embarqués, nous pensons également que cette ingénierie doit être interdisciplinaire, c'est à dire qu'il est nécessaire pour son fonctionnement d'établir des fondements théoriques communs aux différents profils développant les systèmes embarqués. L'utilisation d'outils dédiés à chaque discipline et les glissements sémantiques qui ont lieu lors du passage d'une discipline à l'autre sont des freins pour développer efficacement des systèmes embarqués.

Ainsi, nous aurons comme première perspective l'*interdisciplinarié des métiers* autour de l'ingénierie des systèmes embarqués.

Gestion de la sous-traitance. Si Gérard [Gérard 2000] ne semble rien présumer de la concurrence dans l'ingénierie des systèmes embarqués, il souligne néanmoins, dans le domaine de l'automobile, le fort besoin de gérer les relations de sous-traitance, entre fournisseurs et donneurs d'ordre [Gérard 2000]. Sangiovanni-Vincentelli et al. [Sangiovanni-Vincentelli *et al.* 2004] traitent également le besoin de gérer la sous-traitance pour l'ingénierie des systèmes embarqués. Les auteurs parlent d'horizontalisation de l'industrie de l'électronique. La prise en compte de la gestion de la sous-traitance constitue un problème majeur [Sangiovanni-Vincentelli *et al.* 2004].

La *gestion de la sous-traitance* est, selon nous, orthogonale à *l'interdisciplinarité des métiers*. La sous-traitance est motivée par le besoin de se recentrer sur le cœur de métier alors que l'interdisciplinarité est motivé par les spécificités intrinsèques des systèmes embarqués – l'extensibilité et la taille. L'impact de la sous-traitance sur les processus de développement des systèmes embarqués divergent par rapport à ceux de l'hétérogénéité et la maitrise de la taille des équipes. Ainsi, cette perspective se diffère de par son origine et son impact des autres perspectives étudiées.

Mesure et pilotage de l'avancement du développement. L'ingénierie des systèmes embarqués est perçue comme une ingénierie concurrente et distribuée. Les visions de Broy et de Biehl semblent une nouvelle fois diverger sur ces aspects. Pour Broy [Broy 2006a], plusieurs activités peuvent être réalisées simultanément et les différents développements doivent être synchronisés et assemblés lors de jalons, afin d'assurer que le développement global suit la bonne direction. L'idée d'interaction est également reprise dans Biehl [Biehl 2010], bien que sa perception différente de la pluridisciplinarité de l'ingénierie laisse imaginer une réalisation différente de la concurrence des développements. Le terme « interaction » employé par Biehl réfère à la « concurrence » introduite par Broy [Broy 2006a]. Broy introduit néanmoins une seconde dimension qui est la distribution de l'ingénierie, laissant entendre que la seule séparation des développements selon les équipes n'est pas suffisante et qu'il est également envisageable de paralléliser les équipes d'une même discipline. De plus, Broy introduit un rôle supplémentaire, qui est celui de l'orchestrateur, dont le rôle et de planifier les interactions afin de faciliter les phases d'intégration [Broy 2006a]. Biehl ne semble rien présumer de cette distribution.

Broy identifie l'ingénierie concurrente et distribuée comme un besoin de l'ingénierie des systèmes embarqués [Broy 2006a]. L'ingénierie concurrente guide selon lui le développement. Cette concurrence se traduit aux travers d'activités réalisées simultanément et le développement

est jalonné, jalons orchestrés et planifiés au cours desquels des activités d'intégration sont réalisées afin de coordonner les différents développements.

Deux perspectives orthogonales se distinguent dans le besoin identifié par Broy. D'une part, le besoin de distribuer les développements, ce qui réfère à l'*interdisciplinarité des métiers* et d'autre part le besoin de maitriser la concurrence des développement. Maitriser la concurrence dans de grands développements et de grandes équipes nécessite de pouvoir mesurer l'avancement et de le coordonner. Ainsi, la *mesure de l'avancement du développement* constitue une perspective orthogonale à part entière qu'il est nécessaire de pouvoir explorer indépendamment des deux autres.

Rapidité du temps de développement et maitrise des coûts. Le développement rapide est discuté dans [Gérard 2000, Broy 2006b]. Les cycles de développement tendent à être réduits dans les différents secteurs : trois ans dans le secteur automobile [Gérard 2000] en 2000 et deux ans dans la téléphonie mobile [Strategy Analytics 2012] en 2012.

La nécessité de développer rapidement est motivée par plusieurs aspects de l'ingénierie des systèmes embarqués. D'une part, le coût de développement, difficilement quantifiable aujourd'hui, selon Broy [Broy 2006a]. Le coût de développement impose de réduire les temps de développement. D'autre part, les secteurs d'ingénierie sont fortement contraints par le temps de mise sur le marché [Gérard 2000].

L'ingénierie des systèmes embarqués est soumise à forte concurrence, comme nous pouvons le voir dans le secteur de l'automobile [Gérard 2000] ou de la téléphonie mobile [Gartner 2013, Strategy Analytics 2012]. Cette concurrence impose aux constructeurs de réduire les temps des cycles de développement afin de sortir des produits innovants, personnalisés et conformes aux attentes des utilisateurs [Gérard 2000].

Enfin, un dernier aspect concerne la variabilité des systèmes, identifiée par Gérard [Gérard 2000]. Présentées sous l'angle du secteur automobile, les spécifications doivent être flexibles, extensibles, et modulaires. Ainsi, la prise en compte de la variabilité est difficile sur des cycles de développement longs.

Ces différents aspects motivent aujourd'hui le besoin de réduire les temps des cycles de développement. Gérard met toutefois en garde sur le risque d'impacter la qualité des produits en tentant de raccourcir les temps de cycles de développement [Gérard 2000].

La *rapidité du temps de développement* est fortement motivée par le besoin de réduire les coûts de développement. Elle constitue une perspective de notre taxonomie.

Explicabilité et certification des développements. La certification des systèmes est discutée dans [Broy 2006b]. Afin d'assurer la qualité d'un produit – notamment dans le cas de systèmes critiques mettant en péril la sécurité et pouvant mener à des situations catastrophiques – l'ingénierie des systèmes embarqués requiert la certification des systèmes réalisés. Par exemple, le secteur automobile possède la certification ISO 26262 [ISO 2009], le secteur avionique les normes DO-178B et DO-178C [Zoughbi *et al.* 2011]. Sangiovanni-Vincentelli ne partage pas cette opinion vis-à-vis de la certification, la qualifiant de « fardeau » pour les processus de développement. Il dénonce le manque de confiance que l'on peut avoir dans les autorités de certification et critique le fait que le processus de certification relate uniquement du processus de développement et non de la qualité du logiciel embarqué développé [Sangiovanni-Vincentelli 2007].

Le besoin de documentation est discuté dans [Biehl 2010]. Biehl souligne l'intérêt d'une documentation précisant les choix d'ingénierie effectués – en particulier pour les systèmes critiques – ou plutôt l'impact qu'aurait le manque de documentation : délai de développement, dépassement de coûts, etc. [Biehl 2010]. Par ailleurs, la documentation selon Biehl a plusieurs buts. D'une part, elle permet d'assister le développeur dans son développement. D'autre part, elle permet la certification des systèmes auprès des autorités compétentes. Cet avis est également partagé par Sindico et al.[Sindico *et al.* 2012]. Les auteurs affirment que la documentation est un besoin fondamental pour l'ingénierie des systèmes embarqués, notamment dans leur secteur d'activité – l'industrie de la défense – où les règlementations impliquent la production de nombreuses documentations, chacune suivant ensuite des processus de relecture pour vérifier leur exactitude, explicabilité et leur cohérence. Sindico rejoint donc Biehl sur le besoin de documentation dans un souci de certification, soulignant l'aspect clarté de la documentation. Gérard a également justifié le même besoin de clarté dans la documentation produite, dans un contexte où la documentation relève de la transmission des spécificités depuis le constructeur automobile vers le sous-traitant, cette documentation des spécificités doit être non ambigüe [Gérard 2000].

Dans chacun des travaux, l'intérêt de la documentation est donc révélée pour améliorer le développement. Pour Biehl et Sindico, il s'agit en plus d'assurer la certification du système produit [Biehl 2010, Sindico *et al.* 2012]. Quant à Gérard, il s'agit principalement d'un moyen d'échange clair et non ambigü entre les différentes parties prenantes de la conception d'un système [Gérard 2000]. L'*explicabilité et la certification des développements* constituent selon nous une seule perspective à explorer. Biehl [Biehl 2010] et Sindico et al. identifient le besoin d'établir des spécifications claires dans un souci de certification. Nous pensons que la certification ne va pas sans l'explicabilité des développements. Ainsi, nous rassemblons les deux sous l'unique perspective de l'*explicabilité et la certification des développements*.

Synthèse des pensées des auteurs autour des perspectives

Cette partie présente la synthèse des pensées des différents auteurs cités dans la littérature autour des six axes que nous avons définis. La table 2 résume ces différentes pensées. Parmi les difficultés spécifiques à l'ingénierie des systèmes embarqués, la pluridisciplinarité des équipes et les contraintes en termes de documentation et de certifications sont les plus importantes. La pluridisciplinarité implique des contraintes spécifiques d'orchestration et de synchronisation. Ces contraintes traduisent un besoin fort de définir une *gestion de projet* afin d'accompagner le chef de projet dans la mesure et le pilotage efficace des projets.

Travaux existants	***Perspectives étudiées***				
	Interdisciplinarité des métiers	Gestion de la sous-traitance	Rapidité du temps de développement & maitrise des coûts	Explicabilité et certification des développements	Mesure & pilotage de l'avancement du développement
[Gérard 2000]	▸ **P**luridisciplinaire	▸ **G**estion sous-traitance	▸ **C**ycles réduits ▸ **T**TM ▸ **C**oncurrence ▸ **V**ariabilité	▸ **D**ocumentation pour la transmission de spécifications au constructeur	
[Sangiovanni-Vincentelli 2004; Sangiovanni-Vincentelli 2007]	▸ **P**luridisciplinaire	▸ **G**estion sous-traitance ▸ **H**orizontalisation de l'industrie		▸ **C**ertification du processus et non du produit	
[Broy 2006a; Broy 2006b]	▸ **I**nterdisciplinaire		▸ **R**éduction du coût de développement	▸ **D**ocumentation pour assurer la qualité du système	▸ **S**ynchronisation d'activités parallèles ▸ **O**rcherstrateur
[Henzinger and Sifakis 2007]	▸ **P**luridisciplinaire				
[Biehl 2010, Biehl 2011, Törngren 2013b]	▸ **M**ultidisciplinaire			▸ **D**ocumentation pour assister le développeur et pour certifier le système	▸ **I**ntéractions entre équipes de disciplines différentes

TABLE 2 – Synthèse des pensées des auteurs sur les spécificités de l'ingénierie systèmes embarqués

Synthèse du chapitre

En étudiant différents auteurs, nous avons pu identifier un certain nombre de spécificités des systèmes embarqués et de leur ingénierie. Les différentes spécificités des systèmes traduisent une complexité de développement accrue par rapport aux systèmes purement logiciels. Cette complexité est majoritairement due au caractère temps-réel impliqué par l'instrumentation du monde physique et l'exécution sur une plate-forme aux ressources limitées et apportant de nombreuses contraintes (début, latence, vitesse des processeurs, etc.) sur le système (fiabilité, temps de réponse, etc.).

Du côté de leur ingénierie, cette dernière est fortement contrainte par une pression forte du marché obligeant les constructeurs à réduire leur temps de développement tout en garantissant un niveau suffisant de qualité et de fiabilité. À ce problème s'ajoutent ceux de l'environnement de développement lui-même, pluridisciplinaire dans lequel la documentation et la certification sont preuves de robustesse et garantissent la sureté de fonctionnement. Ces différents aspects traduisent un besoin fort de définir une gestion de projet adaptée facilitant la mesure et le pilotage du développement par le chef de projet.

Avant d'entamer le chapitre 4 dans lequel nous comparons les méthodes actuelles afin d'en dégager les forces et faiblesses et de nous permettre de positionner les contributions que nous avons apportées, il est nécessaire de quitter le monde des systèmes embarqués pour découvrir celui de l'ingénierie des méthodes et des modèles. Cette découverte au travers du chapitre 3 nous permettra de définir précisément ce qu'est une méthode de développement, quels sont ses contours et quelles sont les techniques et les langages permettant de formaliser une méthode.

Chapitre 3

L'ingénierie des méthodes et des modèles

Liste des sections

Contrairement aux communautés des systèmes logiciels et des systèmes d'information, la communauté des systèmes embarqués utilise les termes « méthode » ou « méthodologie » [1] de manière très subjective. Ces termes apparaissent à tous les niveaux et ne possèdent pas le même sens. Ainsi, il n'existe pas de consensus dans le domaine des systèmes embarqués sur ce qu'est une méthode de développement. Les termes « méthode » ou « méthodologie » réfèrent selon le cas à un enchainement d'étapes pour arriver à un résultat, à une technologie particulière mise en œuvre, à une utilisation d'outils. L'ingénierie des systèmes embarqués ne possède pas de lien fort avec celle des méthodes. Parmi les milliers de méthodes de développement de systèmes embarqués que nous trouvons dans la littérature, peu d'entre elles peuvent être réellement qualifiées de méthodes. S'il n'y a pas dans le monde des systèmes embarqués de consensus et ou définition acceptée d'une méthode de développement, ces concepts sont par contre bien stabilisés dans le monde de l'ingénierie des systèmes logiciels et des systèmes d'information.

Avant de pouvoir formaliser une méthode de développement de systèmes embarqués, ce chapitre permet d'identifier les contours d'une méthode de développement et d'introduire les concepts nécessaires à sa formalisation. Nous verrons, aux travers des différents modèles de processus mis en place depuis quarante ans quelles sont les principales caractéristiques de ces derniers et en quoi ils sont aujourd'hui nécessaires pour le développement de systèmes logiciels. Ce chapitre est structuré en trois sections : la première section présente l'ingénierie des méthodes ; la seconde détaille les concepts de processus et de modèle de processus ; la troisième présente les techniques de l'ingénierie des modèles permettant la modélisation des modèles de processus.

1. La communauté des systèmes embarqués utilise indifféremment l'un ou l'autre terme.

1 L'ingénierie des méthodes

Les méthodes sont une réponse à la complexité des systèmes logiciels [Booch 1994]. Nous pouvons trouver dans la littérature de l'ingénierie des méthodes et du logiciel plusieurs définitions du terme « méthode ». La première caractérisation d'une méthode est celle de Seligmann, Wijers et Sol [Seligmann *et al.* 1989]. Une méthode est caractérisée selon quatre manières : la manière de penser (« *the way of thinking* »), d'organiser (« *the way of organizing* »), de modéliser (« *the way of modelling* ») et de supporter (« *the way of supporting* »). La manière de penser fait référence à la démarche, c'est-à-dire à l'organisation et au guidage des activités de modélisation, de description et de réalisation. La manière de modéliser décrit le type des modèles, leurs constructions et leurs relations. Le support se traduit par l'usage de techniques et d'outils automatiques pour la description des modèles. La manière d'organiser se décline en la manière de travailler (« *way of working* ») et de superviser le travail (« *way of control* »).

Booch [Booch 1994] propose une seconde définition d'une méthode :

> « Une méthode d'ingénierie des systèmes est un processus rigoureux permettant de générer un ensemble de modèles qui décrivent divers aspects d'un logiciel en cours de construction en utilisant une certaine notation bien définie. » [Booch 1994]

Dans cette définition, les termes « processus rigoureux » désignent un ensemble d'activités menant à la construction ordonnée d'un système de *modèles*. La *notation* est le langage d'expression de chaque modèle. Booch souligne le beoin de posséder des notations bien définies et expressives, pour permettre de communiquer sans ambiguïté, de réduire l'effort intellectuel de décryptage et de concentrer cet effort sur le véritable besoin [Booch 1994].

Une troisième définition des méthodes dans le contexte des systèmes d'information est proposée par Harmsen [Harmsen 1997] :

> « Une méthode pour l'ingénierie des systèmes d'information est une collection intégrée de procédures, techniques, descriptions de produits et outils pour un support performant, efficace et consistant des processus d'ingénierie de systèmes d'information. » [Harmsen 1997]

Dans cette définition, nous retrouvons les différentes manières décrites par Seligmann, Wijers et Sol. La collection intégrée de procédures désigne la démarche, les techniques et descriptions de produits réfèrent à la manière d'organiser et de modéliser. On y retrouve également le support du langage et du processus par l'outillage. Pour la suite, nous nous appuierons sur les définitions précédentes afin de donner notre propre définition d'une méthode.

Définition 1 (Méthode) Une *méthode* est constituée d'une démarche guidée composée d'un ensemble d'activités ordonnées dont le suivi permet de produire un résultat et d'un ensemble de langages permettant de décrire le résultat produit.

Définition 2 (Méthode formalisée) Une méthode *formalisée* est une méthode composée de langages formalisés et d'une démarche modélisée (*modèle de processus*). Un *langage formalisé* est un langage défini par une syntaxe *abstraite* (le méta-modèle), une syntaxe *concrète* (la notation) et par une sémantique (le dictionnaire des concepts).

Définition 3 (Méthode formalisée et outillée) Une méthode *outillée* est une méthode définissant un ensemble d'outils supportant le langage de la méthode et éventuellement le modèle de processus et dont l'usage permet d'assister, de guider et d'automatiser certaines activités menant à la production d'un résultat.

2 Les modèles de processus

Le terme « processus » désigne « une approche systématique pour la production d'un produit ou l'accomplissement d'une tâche » [Osterweil 1987]. Le terme processus s'applique à de nombreux domaines, tels que que les processus d'entreprise, ou les processus de développement logiciels. Ces derniers ont connu une forte croissance dès lors que la communauté du logiciel a commencé à attacher aux processus de développement logiciel la même importance qu'au logiciel lui-même [Osterweil 1987]. La vision du développement logiciel a progressivement évolué, de la création simple d'outils et d'applications vers un effort créatif et collectif, laissant apparaître toute la complexité du processus de développement [Fuggetta 2000]. Dans cette section, nous nous intéressons à la modélisation de ces processus.

2.1 Définition des modèles de processus

Une modèle de processus est une représentation d'un processus. Finkelstein et al. définissent un modèle de processus comme « la description d'un processus exprimé dans un langage de modélisation adapté » [Finkelstein *et al.* 1994]. Le processus est ainsi considéré comme un modèle et possède à l'instar du système à produire, son cycle de vie, allant de sa conception et sa modélisation à son exécution et à son contrôle [Rafique-Golra 2014]. Dans le cadre des processus métiers, Godart [Godart & Perrin 2009] définit un modèle de processus :

> « Représentation d'un processus métier qui supporte des manipulations automatiques par un système de gestion de workflow. » [Godart & Perrin 2009]

Godart définit également un système de gestion de workflow de la façon suivante :

> « Un système de gestion de workflow est un système qui définit, crée et gère l'exécution de workflows par l'utilisation de logiciels capable d'interpréter les définitions de processus, d'interagir avec les participants et, lorsque cela est requis, d'invoquer les outils et les applications. » [Godart & Perrin 2009]

Différentes dimensions se dégagent de ces définitions. Un modèle de processus est un modèle conforme à un langage de modélisation de processus permettant de définir l'organisation d'activités manuelles et automatiques. Un modèle de processus définit des rôles et des participants permettant d'accomplir les différentes activités. Les activités automatiques peuvent être supportées par différents outils. Enfin, un modèle de processus peut être associé à un système de gestion de workflow pour assurer et permettre le contrôle de son exécution.

Les modèles de processus ont été très vite appliqués par la communauté du logiciel pour le développement logiciel. Ils définissent un moyen rigoureux d'arriver à la production d'un système logiciel. De nombreux modèles de processus ont été proposés au cours du temps, chacun possédant ses caractéristiques, ses avantages et ses inconvénients. Nous citons dans cette partie quelques principaux modèles de processus.

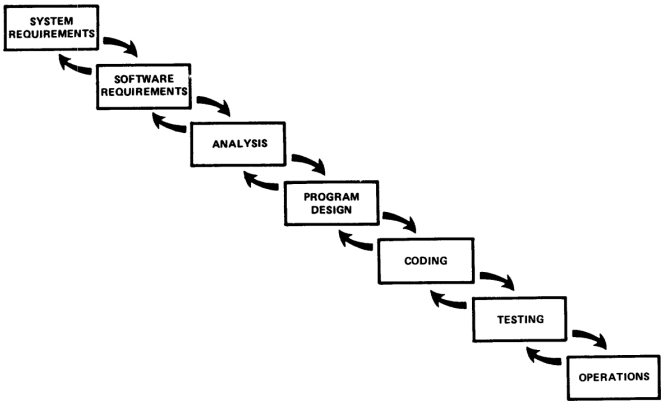


FIGURE 6 – Modèle en cascade originel avec retours en arrière [Royce 1970]

2.2 Les modèles linéaires.

Les modèles linéaires sont apparus dès le début des années 1970 [Royce 1970]. Ils restent aujourd'hui très populaires pour la production de logiciels. Ils sont constitués d'un ensemble d'étapes ordonnées qui sont réalisées séquentiellement. Chaque étape doit se conclure sur la production d'un document permettant de valider l'étape. Cette approche présente l'avantage de forcer la documentation du développement. Les deux modèles les plus connus sont le modèle *en cascade* proposé par Royce [Royce 1970] et le très populaire modèle *en V*.

Modèle en cascade [Royce 1970]. Le modèle originel proposé par Royce est illustré par la figure 6. Il est composé de sept étapes permettant de développer un logiciel. Il est majoritairement guidé par la production de documents à chaque fois qu'une étape se termine. À l'opposé des méthodes Agiles [Beck *et al.* 2001, Highsmith & Fowler 2001], Royce insiste sur la nécessité de produire un très grand nombre de documents [Royce 1970]. Par exemple, durant l'étape de conception du programme, une spécification de la conception du programme est produite, ainsi qu'un plan de test est conçu qui sera utilisé durant l'étape de test pour obtenir les résultats des tests. Royce suggère également une dixième étape, de conception préliminaire du programme venant s'intercaler entre les phases de recueil du besoin et d'analyse. Cette étape permet de s'assurer que le programme fonctionnera conformément aux contraintes techniques imposées.

Le modèle communément appelé « en cascade » que nous connaissons aujourd'hui qui est constitué d'un strict enchainement d'activités de recueil du besoin, d'analyse de conception et de développement diffère du modèle originellement proposé par Royce [Larman & Basili 2003]. Royce préconisait déjà à l'époque différentes caractéristiques aujourd'hui appliquées dans les modèles évolutifs et les méthodes Agiles. Le retour arrière fut proposé par Royce, tout en précisant qu'il doit se limiter à un retour sur l'étape directement en amont. Royce préconisait également que le développement soit réalisé en deux passes, dans le cas des processus incluant de nouveaux éléments et de nouveaux facteurs inconnus pouvant augmenter le risque de développement [Royce 1970, Larman & Basili 2003]. Les deux passes réalisées laissent présupposer d'un développement (bien que limité) incrémental. En tout dernier lieu, Royce encourage l'implication du client dans le processus pouvant apporter son jugement sur le développement en cours. Cette implication s'effectue dès la phase de conception préliminaire du programme. L'ajout

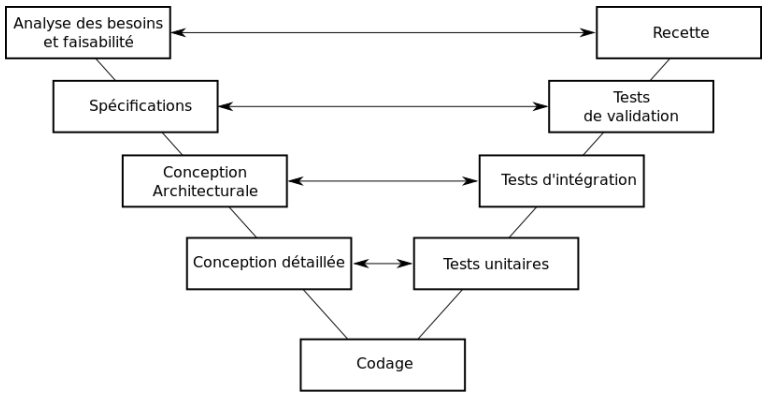


FIGURE 7 – Le modèle en spirale

de cette phase, accompagnée du retour et des impressions du client permettent de rediriger le développement dans les phases en aval.

Modèle en V [McDermid & Ripken 1983]. Le modèle en V [McDermid & Ripken 1983] est illustré par la figure 7. Il est composé de neuf phases et met en vis-à-vis les différentes phases de réalisation (analyse des besoins et faisabilité, spécifications, conception architectural, conception détaillée) et les recettes et tests associés (recette, tests d'acceptation, tests d'intégration, tests unitaires) à chacune des phases. L'apposition des tests en vis-à-vis permet de déceler très tôt les erreurs et de réduire le retour en arrière. Le processus se déroule chronologiquement selon les neufs phases et à l'issue de chaque phase, le test associé est créé. Ce modèle est parmi les plus simples et les plus populaires.

2.3 Les modèles itératifs et évolutifs

Les modèles linéaires présentent plusieurs inconvénients. Ils ne permettent pour la plupart pas de retour en arrière possible. Ils sont adaptés à des projets dont les exigences sont connus dès le départ et non sujettes à modification. Bien que très populaires, ces modèles souffrent d'un manque de flexibilité et la production de documents et de tests à chaque étape les rendent irréalisables et inadaptés à de nombreux projets [Gilb 1985]. Notamment, le modèle ne fait pas entrer le client dans les phases de production et le produit final est la première chose que voit le client. Tous les processus ne sont pas séquentiels, c'est-à-dire ne présentent pas un enchainement d'étapes dans un seul sens. Un processus non séquentiel est dit itératif. Un développement itératif est défini par Benediktsson et al. ainsi [Benediktsson *et al.* 2003] :

> « Le développement itératif est une stratégie pour développer des systèmes permettant de retravailler sur une partie du système dans le but de supprimer des erreurs ou de réaliser des améliorations basées sur des retours utilisateur. » [Benediktsson *et al.* 2003]

Dans [Graf 1999], Graf propose la définition suivante :

> « Un processus itératif est un processus pour obtenir un résultat désiré au moyen de cycles d'opérations répétés. » [Graf 1999]

Un processus itératif permet de concevoir un résultat graduellement au moyen d'itérations successives. Les mêmes étapes sont répétées, et chaque itération peut se conclure sur le retour et l'impression de l'utilisateur final pouvant juger les ajouts effectués sur le produit durant l'itération. On associe très souvent dans la littérature le concept d'*itération sur le processus* à celui d'*incrément sur le produit* en cours de développement. Nous adoptons la définition de Graham [Graham 1989] – reprise par Benediktsson [Benediktsson *et al.* 2003] :

> « Un incrément est une unité fonctionnelle autonome d'un logiciel avec tout le matériel de support telles que les documentations des spécifications et de la conception, les manuels utilisateur et les formations. » [Graham 1989]

Les modèles itératifs et évolutifs [Gilb 1976, Larman & Basili 2003] mettent l'accent sur le développement rapide et la participation du client dans le processus pour obtenir ses impressions et ses réactions afin de pouvoir les utiliser dans une itération suivante. Ils prennent en compte la nature changeant du projet et des besoins des clients et permet de découper un projet en incréments constituant des sous-systèmes indépendants nécessitant une phase d'intégration dans le produit final. Le développement peut alors commencer avant que toutes les exigences ne soient définies, permettant ainsi de réduire le temps de mise au marché [2].

Modèle en spirale [Boehm 1988]. Le modèle en spirale est un modèle de processus de développement logiciel proposé en 1988 par Boehm basé sur son expérience acquise sur l'utilisation du modèle en cascade [Boehm 1988]. Le modèle en spirale est un modèle itératif qui inclut des phases d'analyse de risque. La figure 8 illustre le modèle. La dimension radiale représente le coût de développement de la tâche en cours, cumulé aux coûts de toutes les phases ayant déjà été réalisées. La dimension angulaire quant à elle reflète la progression faite dans l'accomplissement des tâches définies dans un cycle de la spirale. Ce modèle est itératif et incrémental, chaque

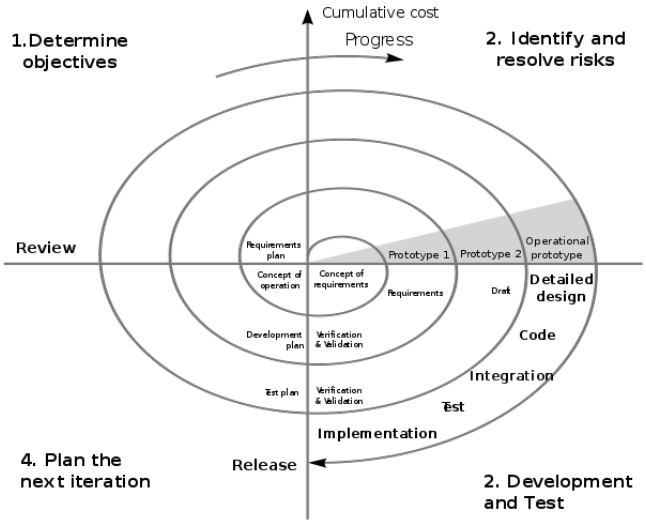


FIGURE 8 – Le modèle en spirale [Boehm 1988]

2. En anglais, Time-to-Market (TTM)

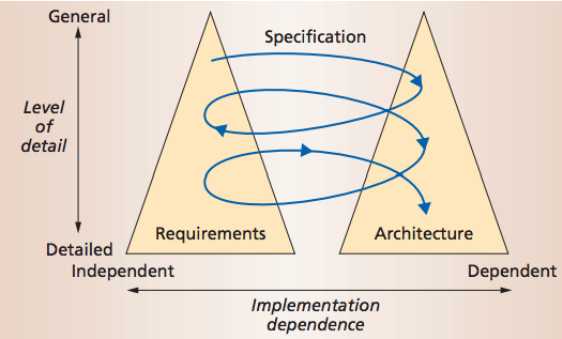


FIGURE 9 – Le modèle « Twin Peaks » [Nuseibeh 2001]

cycle est scindé en différentes tâches (identification des risques, développement et test, planification de l'itération suivante et détermination des objectifs) formant un nouvel incrément sur le développement. Le modèle formé représente alors une spirale imparfaite (le coût de chaque étape et la progression effectuée varient en fonction du cycle réalisé). D'autres modèles se sont par la suite basés sur ce dernier, comme c'est le cas du *twin-peaks model* [Nuseibeh 2001].

Modèle « Twin Peaks » [Nuseibeh 2001]. Le modèle « Twin Peaks » [Nuseibeh 2001] est une adaptation du modèle en spirale et a été proposé par Nuseibeh en 2001 pour répondre au manque de souplesse des modèles en cascades vis-à-vis des spécifications changeantes et évolutives d'un projet. Le modèle s'appuie sur le fait que les activités de spécification et d'implémentation sont réalisées de façon concurrente et entremêlée. Ce modèle repose sur différents principes : l'utilisation de logiciels pris sur l'étagère, communément appelé COTS [3] ; la réactivité aux changements rapides des spécification ; la construction modulaire d'un logiciel par incréments successifs et enfin, la plus grande particularité, l'entremêlement des phases de spécification et de développement illustré par la figure 9.

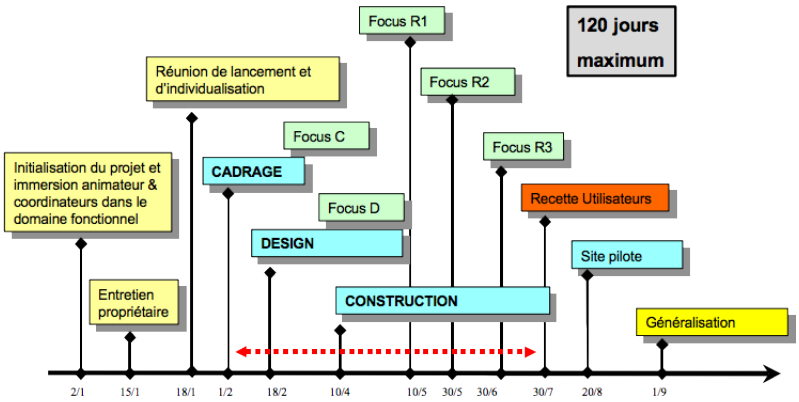


FIGURE 10 – Principales étapes d'un projet à 120 jours [Vickoff 2000]

3. En anglais, Commercial Off-The-Shelf (COTS).

Développement Rapide d'Application (RAD) [Martin 1991]. Rapid Development Application (RAD) [Martin 1991, Vickoff 2000] est un modèle basé sur le cycle en spirale de Boehm. Ses objectifs sont le développement rapide de livrables, la haute qualité des systèmes et la réduction des coûts. Martin définit la qualité d'un système comme la « satisfaction des besoins de l'utilisateur dans la mesure du possible au moment où le système devient opérationnel » [Martin 1991]. RAD est basé sur cinq phases qui sont l'initialisation, le cadrage, la conception, la construction et la finalisation (cf. fig. 10). Les principes de RAD ont largement inspiré les méthodes Agiles. RAD repose sur un modèle *semi-itératif* (premières phases de cadrage et de conception classiques, suivies d'une phase de construction avec des itérations courtes) et incrémental pour la construction de prototypes et implique le client final. L'usage de prototypes permet à la fois d'impliquer l'utilisateur, d'évaluer et de tester de nouvelles solutions. RAD aborde les différents aspects tels que les coûts, les utilisateurs ou encore la durée des projets.

Modèle en Y [Luiz Fernando Capretz 2005]. Le modèle en Y [Luiz Fernando Capretz 2005] a été inventé pour répondre aux affirmations suivantes. D'une part, le développement logiciel basé sur les composants promeut la réutilisation, l'amélioration de la qualité et de la productivité. D'autre part l'ingénierie du logiciel se base sur les aspects itératif et incrémental. Le modèle en Y fut proposé pour répondre aux besoins de réutilisation dans les phases de conception de logiciels basés sur les composants. Il met principalement l'accent sur la création, l'évolution et la production de composants s'avérant utiles pour les futurs projets logiciels. Cela amène deux nouvelles notions, qui sont la conception de librairies de composants réutilisables, et l'héritage de composants, notion déjà connue dans le monde de l'objet. La figure 11 illustre le modèle, constitué de deux branches, formant la lettre « Y ». Il apporte une séparation nette entre les aspects fonctionnels et techniques, et inclut au sein même de son processus l'aspect réutilisation de composant (les étapes de frameworking et d'assemblage).

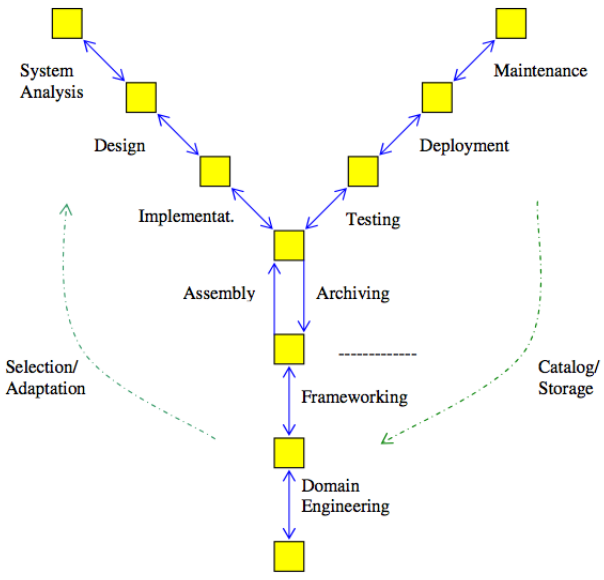


FIGURE 11 – Le modèle en Y [Luiz Fernando Capretz 2005]

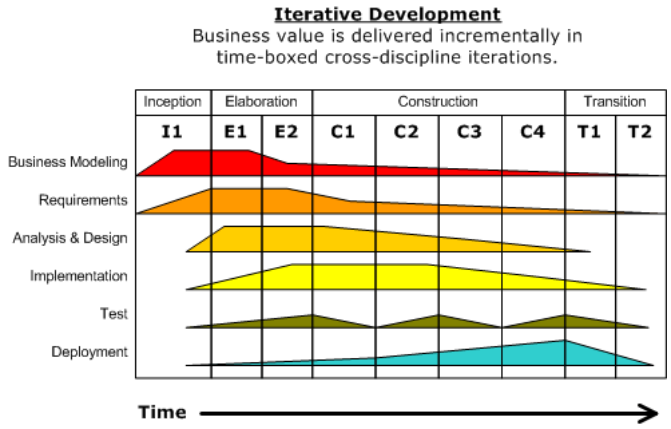


FIGURE 12 – Processus itératif dans USDP [Ivar Jacobson *et al.* 1999]

2.4 Les processus unifiés

Les processus unifiés réfèrent à une méthode de développement générique et sont basés sur les modèles UML [4]. Ils arborent une structure itérative et incrémentale. La principale caractéristique est l'utilisation des diagrammes de cas d'utilisation apportés par Jacobson afin de guider les itérations. De nombreuses implémentations ont été proposées, telles que Unified Software Development Process (USDP) et Rational Unified Process (RUP).

USDP [Ivar Jacobson *et al.* 1999]. USDP [Ivar Jacobson *et al.* 1999] est un modèle décomposé en quatre phases qui sont le commencement, l'élaboration, la construction et la transition (cf. fig. 12). Chacune peut être scindée en plusieurs itérations. L'aspect itératif et incrémental est ainsi la première caractéristique du processus. Le processus est centré sur les cas d'utilisation modélisant les besoins fonctionnels du système. Le contenu d'une itération est définie en termes de cas d'utilisation qu'elle doit couvrir. Enfin, USDP s'appuie sur une définition et une analyse du risque tout au long du processus. Les itérations sont organisées et sélectionnées afin de couvrir les plus grands risques dès les premières phases du processus.

RUP [IBM 2003]. RUP [IBM 2003] est l'une des plus célèbres implémentations des préceptes du processus unifié. Il s'agit d'un modèle visant à guider l'utilisateur tout au long du développement logiciel. Il a été proposé par Rational Software (IBM) et fut intégré dans la suite logicielle de développement d'outils de Rational en 1998. Cette approche ajoute quelques bonnes pratiques de développement aux pratiques définies par USDP. RUP indique comment modéliser le système en capturant à la fois le modèle statique (la structure) et le modèle dynamique (le comportement) des composants. Cette abstraction permet d'identifier les liaisons entre les différents composants du système. Cette modélisation est basée sur les concepts issus de UML. Enfin, RUP rend possible le changement. Il décrit comment surveiller, tracer et contrôler les changements. Il permet également d'établir un espace de travail pour chaque développeur lui permettant de récupérer une copie du travail courant et lui donnant la possibilité de réaliser ses propres modifications. RUP fournit également des artefacts de gestion de projet.

4. En anglais, UML.

2.5 Synthèse sur les modèles de processus

Les modèles de processus présentés possèdent différentes caractéristiques les rendant plus adaptés à un projet particulier. Par exemple, les modèles en cascade sont plus adaptés à des projets dont les spécifications ne varient pas ou peu et où la production de documentation est le point fondamental [Sindico *et al.* 2012]. À l'inverse, pour un projet où les spécifications varient beaucoup, des modèles évolutifs et les méthodes Agiles sont plus facilement applicables.

Différentes caractéristiques communes à plusieurs modèles sont mises en avant dans les modèles présentés. Les caractères itératif et incrémental sont très largement mis en avant dans la plupart des processus. Viennent ensuite l'implication de l'utilisateur final, le développement rapide, et le prototypage, la concurrence des développements et la réutilisation de composants et de COTS qui constituent également des caractéristiques des processus fortement appréciées. La section suivante aborde les aspects modélisation et exécution de ces modèles de processus.

3 Modélisation et exécution de processus

La modélisation des processus de développement logiciels réfère à la définition du processus en tant que modèle et des différents supports permettant sa modélisation et son exécution [Curtis *et al.* 1992, Acuña & Ferré 2001]. La modélisation de ces processus offre de nombreux avantages tels que la simplification de la compréhension et de la communication au travers des modèles, le support et le contrôle des processus ou bien encore la possibilité d'automatiser l'exécution du processus permettant d'accomplir des tâches automatiques [Curtis *et al.* 1992].

3.1 Les trois niveaux de modélisation

La modélisation des processus de développement logiciel suit le même cycle que celui des logiciels eux-même. Rolland [Rolland 1993] présente les trois niveaux d'abstraction des processus de modélisation (cf. fig. 13 sur laquelle nous avons rajouté les trois niveaux de modélisation). Le niveau M1 représente le modèle de processus. Le modèle de processus décrit un processus en M0. Le processus est alors une instance du modèle. Il représente un cas d'exécution possible du modèle de processus et les données manipulées [Godart & Perrin 2009]. Au niveau M2, un méta-modèle de processus permet la modélisation des modèles de processus. Différents méta-modèles ont été proposés et ciblent différent domaines et types d'orientation de processus (activités, produits, décision, contexte, stratégie) [Rolland 1993]. Par exemple, Software & Systems Process

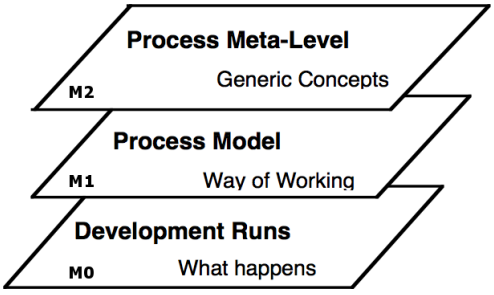


FIGURE 13 – Les trois niveaux d'abstraction des processus de modélisation [Rolland 1993]

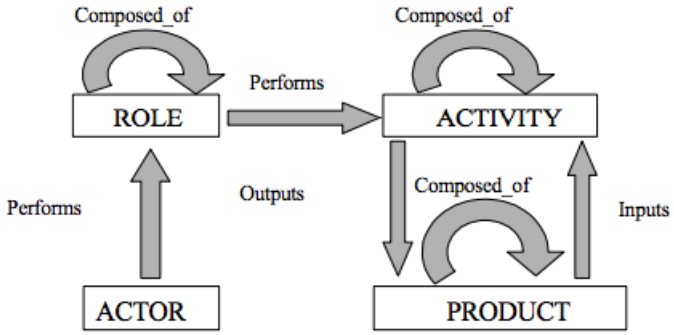


FIGURE 14 – Les concepts élémentaires pour la modélisation des processus orientés activités et produits [Acuña & Ferré 2001]

Engineering Metamodel (SPEM) est un processus orienté activités ciblant la modélisation des processus de développement logiciel. Nous nous intéressons dans le cadre de cette thèse aux méta-modèles de processus orientés activités.

3.2 Les concepts de modélisation

Les modèles de processus orientés activités représentent les activités et leur ordonnancement pour la réalisation d'un produit [Rolland 2005]. Plusieurs travaux ont proposé une taxonomie des concepts permettant la modélisation de tels processus [Curtis *et al.* 1992, Acuña & Ferré 2001, Godart & Perrin 2009]. Godart et Perrin les catégorisent dans trois dimensions : logique (quelles activités ?), organisationnelle (qui exécute ?) et informationnelle (avec quoi ?). La figure 14 présente les concepts élémentaires et leurs relations [Acuña & Ferré 2001].

Activité et tâche. Une activité est une partie d'un processus qui constitue un travail à effectuer pour produire un résultat. L'activité peut être automatique ou manuelle et nécessite des ressources (humain ou ordinateur) pour être réalisée. Elle peut être définie en termes d'états et être accompagnée de pré et de post-conditions sur les produits en entrée et en sortie de l'activité. On distingue généralement plusieurs activités, telles que le sous-processus, l'activité atomique (on part généralement de tâche) ou encore l'activité en boucle. Au niveau M0, une activité est instanciée en une *instance d'activité*.

Produit. Le produit est le résultat d'une activité. Il peut également désigner un support pour l'application du processus. Il peut au même titre que l'activité être défini en termes d'états. Au niveau M0, un produit peut être créé, modifié et supprimé.

Acteur et rôle. Un acteur est une « entité qui exécute un processus » [Godart & Perrin 2009]. Tout comme les activités, nous pouvons distinguer les acteurs humains des ordinateurs. Un acteur joue au sein du processus un rôle ou plusieurs rôles. Chaque rôle confère à l'acteur un ensemble de responsabilités et d'actions possibles sur le processus.

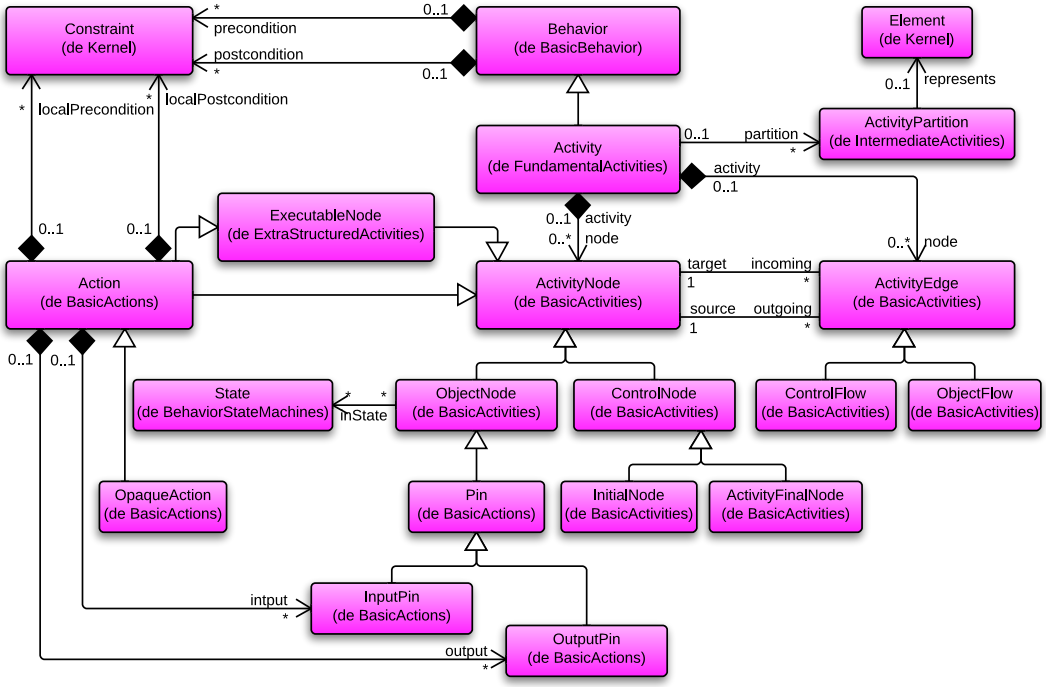


FIGURE 15 – Syntaxe abstraite des activités dans UML

3.3 Méta-modèles de processus orientés activités et produits

Différents méta-modèles de processus orientés activités et produits ont été proposés.

Les diagrammes d'activités et de machine à états UML. Le langage UML [Object Management Group 2010] fournit un ensemble de concepts qui peuvent être utilisés pour modéliser des modèles de processus orientés activités et produits. Cet ensemble de concepts est principalement contenu dans les paquetages UML *Activities*, *Actions* et *Statemachines*. La figure 15 présente la syntaxe abstraite des activités UML. Une activité UML est constituée de nœuds (*ControlNode*, *ObjectNode*), de nœuds exécutables (*Action*, *OpaqueAction*, ...), de flots (*ControlFlow* et *Object Flow*) et de groupes d'activités (*ActivityPartition*, etc.). Une *ActivityPartition* est un groupe permettant de regrouper des nœuds d'activités. Chaque partition référence un *représentant*, un élément UML, représentant habituellement une personne ou une organisation. Les actions opaques permettent de définir une action dans un langage spécifique (langage naturel, Java, C++, ...). Dans les diagrammes d'activités, les objets sont des entrées / sorties d'actions. Un association *inState* permet de définir dans quel(s) état(s) doit se trouver un objet en entrée d'une action et dans quel(s) état(s) il doit en sortir. Les actions sont des *noeuds exécutables*. La notion d'exécutabilité est assez floue dans la spécification de la version 2.4.1 de UML [Object Management Group 2010] et un peu plus détaillée dans la proposition de la version béta de UML 2.5 [Object Management Group 2013] :

> « *Un moteur d'exécution est un outil qui exécute des actions.* » [Object Management Group 2013, p. 458]

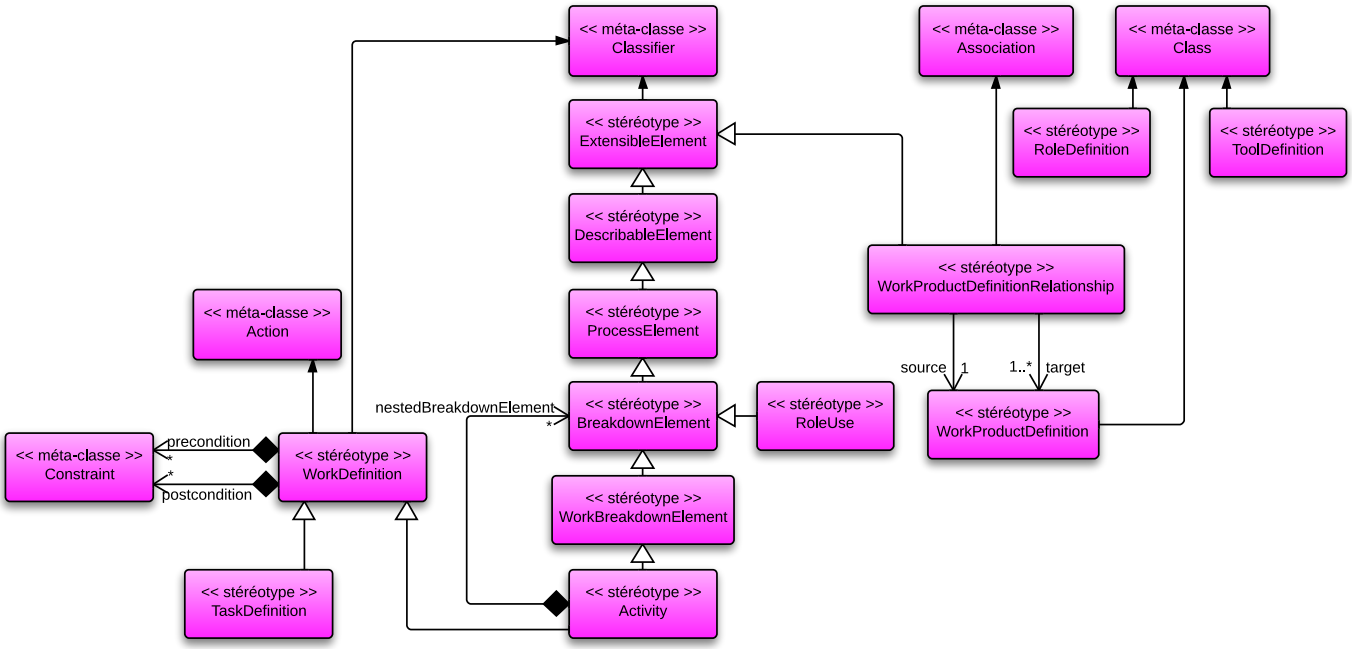


FIGURE 16 – Profil UML de SPEM

Le langage des actions UML permet notamment l'exécution et la simulation de modèles par des outils. Par exemple, [Kirshin *et al.* 2007] a développé un moteur d'exécution de modèles comportementaux UML comme extension de l'outil Rational Software Architect (RSA) et basé sur la plate-forme Eclipse. Il permet de simuler pas à pas ou en continu des diagrammes d'activités et des machines à états avec la possibilité de placer des points d'arrêt.

SPEM. SPEM est un autre langage de modélisation de processus adressant essentiellement les processus de développement dans l'ingénierie logicielle. Il est basé sur la version 2.0 du méta-modèle du Meta-Object Facility (MOF) [OMG 2008] et est proposé à la fois sous la forme d'une extension du méta-modèle UML et d'un profil UML (cf. fig. 16). Il offre beaucoup plus de concepts pour la construction de processus que le diagramme d'activités, avec les concepts de *jalon*, d'*activité*, de *rôle*, de *produits*, etc. Une originalité de SPEM par rapport aux diagrammes d'activités UML est qu'une activité dans SPEM étend à la fois les méta-classes *Action* et *Classifier* d'UML. Ainsi, il est possible d'instancier des activités au niveau M0 ce qui est particulièrement pratique pour exécuter et historiser les activités. En plus d'un guidage précis du processus, il est possible de récupérer une *trace d'exécution* du processus. Initialement, la première version de SPEM présentait un certain nombre de limitations en termes d'exécution, de simulation et de modélisation du comportement. Ces problèmes ont partiellement été traités par la nouvelle version 2.0 du standard, mais les concepts ajoutés restent néanmoins insuffisants, en particulier pour l'exécution d'un processus [Bendraou *et al.* 2007]. Par exemple, à l'instar de UML, SPEM suggère l'utilisation d'un *moteur de workflow* en spécifiant toutefois que pour cela, les modèles SPEM doivent être raccordés à des modèles comportementaux au travers du paquetage *Process Behavior*.

> « *La séparation des modèles SPEM 2.0 et des modèles comportementaux* [...] *permet d'utiliser des moteurs de workflow pour différents types d'approches de modélisation comportementale. En d'autres termes, puisque les modèles SPEM 2.0 peuvent être reliés aux éléments des modèles comportementaux, n'importe quel moteur de workflow créé pour ces modèles peut être utilisé.* » [OMG 2008, p. 149]

Dues aux limitations de SPEM, plusieurs extensions telles que xSPEM [Bendraou *et al.* 2007] ou eSPEM [Ellner *et al.* 2010] ont été proposées. Elles apportent plusieurs nouveaux concepts pour la caractérisation et la surveillance du suivi de projets de différentes tailles et la représentation dynamique d'information, utile pour pouvoir exécuter et organiser le développement.

Notions	**Activités et Machine à états**	**SPEM**
	Activités	
Activité	*Activity*	*Activity*
Tâche	*Activity Node*	*Task Definition Definition*
Pré-condition	*Constraint*	*Constraint*
Post-condition	*Constraint*	*Constraint*
Acteur	*Swimlane*	*Role Definition*
	Produits	
Produit	*Pin*	*Work Product Definition*
Entrant	*Input Pin*	*Parameter Direction Kind*
Sortant	*Output Pin*	*Parameter Direction Kind*
États des produits	association *inState : State[*]*	–
	Exécution	
Exécutabilité	*ExecutableNode*	suggestion
Traces d'exécution	–	Extension de *Classifier*

TABLE 3 – Comparaisons des notions présentes dans UML et dans SPEM

La table 3 synthétise et compare les concepts existants dans les deux langages par rapport aux besoins évoqués en début de section. Les concepts élémentaires pour la modélisation d'activités, de tâches et d'acteurs sont communs aux deux langages. Les deux s'appuient sur le concept de contraintes UML afin de spécifier des pré et post conditions sur les activités et les tâches en termes de produits et d'états des produits. Les produits sont définis dans les deux cas. UML distingue par deux concepts distincts les produits entrants et sortants tandis que SPEM propose une énumération avec trois choix possibles « *in* », « *out* » et « *inout* ». Dans le cas des diagrammes d'activités, chaque objet est un *classifier* pouvant posséder une machine à états. Une association « *inState* » permet d'associer plusieurs états possibles à un produit entrant ou sortant. SPEM n'explicite pas d'état et ne propose pas de concept de modélisation dynamique [OMG 2008, p. 69], mais offre des suggestions de raccordement à des langages prenant en compte cet aspect, notamment les machines à états UML ou bien Business Process Model and Notation (BPMN). En terme d'exécution, les deux proposent seulement des suggestions, dans UML, il s'agit d'un *moteur d'exécution*, dans SPEM, il s'agit d'un *moteur de workflow*. UML, en particulier le paquetage *Actions*, fait un pas de plus en proposant le concept de *nœud d'exécution*. Enfin, SPEM permet d'instancier des *tâches*, puisque les concepts d'*Activity* et de de *Task Definition* étendent le concept de *Classifier* de UML.

Synthèse du chapitre

Ce chapitre introduit l'état de l'art de l'ingénierie dirigée par les méthodes et les modèles. Nous avons tout d'abord donné une définition d'une méthode de développement formalisée. Nous avons présenté les concepts existant pour les processus de développement logiciel et leur modélisation. Nous nous sommes concentrés sur les modèles de processus orientés activités et produits, ainsi que les langages de modélisation (SPEM, UML) de ces processus. Les processus de développement étudiés présentent différentes caractéristiques communes, telles que le besoin de construire par itérations successives produisant une série d'incréments sur le produit conçu, ou encore la conception de prototypes tôt dans le processus et l'implication de l'utilisateur final.

Nous avons étudié dans le chapitre précédent en quoi les systèmes embarqués diffèrent des systèmes logiciels et quelles sont leurs spécificités et celles de leur ingénierie. Afin de proposer une méthode de développement de systèmes embarqués, nous devons établir un pont entre l'ingénierie des systèmes d'information et celle des systèmes embarqués. Pour cela, il est nécessaire d'étudier en quoi les caractéristiques étudiées dans ce chapitre permettent de répondre aux différentes spécificités des systèmes embarqués et de leur ingénierie et de voir comment ces caractéristiques sont prises en compte dans les méthodes actuelles. Cette étude fait l'objet du prochain chapitre.

Chapitre 4

Les méthodes de développement de systèmes embarqués

Liste des sections

Ce chapitre permet d'établir le lien entre l'ingénierie des méthodes et l'ingénierie des systèmes embarqués. Il comporte trois sections. la section 1 identifie dans la littérature les principales caractéristiques des méthodes et présente comment elles sont perçues dans le domaine des systèmes embarqués ; la section 2 établit une étude comparative des différentes méthodes en prenant comme base d'étude la couverture des caractéristiques identifiées dans la section précédente ; dans la section 3, nous synthétisons les différentes forces et faiblesses de ces méthodes et positionnons notre approche par rapport à ces dernières.

1 Caractérisation des méthodes de développement de systèmes embarqués

Cette section présente les caractéristiques que doivent posséder les processus, langages et outils pour le développement des systèmes embarqués. Nous montrons en quoi ces caractéristiques répondent aux spécificités des systèmes embarqués ainsi qu'à leur ingénierie.

1.1 Les caractéristiques des processus

Si de nombreux travaux s'intéressent à caractériser les langages, il n'existe pas de taxonomie aussi complète pour les processus de développement de systèmes embarqués. Nous nous appuierons dans cette partie sur la taxonomie de [Céret *et al.* 2013], généralisée à tous les processus de développement. Cette taxonomie regroupe 34 caractéristiques autour de 6 axes

identifiés : *cycle*, *collaboration*, *artéfacts*, *usage recommandé*, *maturité* et *fléxibilité*. Elle cible tous les processus de développement et pas seulement ceux de l'ingénierie des systèmes embarqués. Plus spécifiques aux systèmes embarqués, Broy identifie également plusieurs aspects des processus [Broy 2006a]. Il ne propose néanmoins pas une taxonomie propre aux systèmes embarqués. En comparant les différentes caractéristiques identifiées par Céret [Céret *et al.* 2013] et Broy [Broy 2006a] avec les spécificités des systèmes embarqués et de leur ingénierie, nous identifions dans cette partie les principales caractéristiques des processus qu'il nous semble important d'adopter pour le développement des systèmes embarqués.

Guidage. Le guidage est une caractéristique discutée dans [Broy 2006a]. Elle est essentielle pour assurer un suivi d'un processus faisant intervenir différentes parties prenantes, avec des compétences diverses et dans un contexte d'une ingénierie concurrente et parallèle [Broy 2006a]. Les systèmes embarqués font intervenir des parties prenantes d'horizons et de compétences différentes. Les développements d'une application et d'une plate-forme étant séparés, il est nécessaire de prévoir des points de synchronisation entre les développements.

De manière générale, le guidage du processus permet d'assurer un bon enchainement des activités. Les différentes spécificités des systèmes embarqués – co-développement logiciel et matériel, systèmes distribués, extensibles et multi-fonctionnelles – rendent le développement de ces systèmes modulaire et implique des équipes pluridisciplinaires devant collaborer pour construire un résultat. Dans ce contexte, le *guidage* du processus est primordial et doit clairement préciser les points de synchronisation entre les différentes parties-prenantes des équipes de développement de systèmes embarqués.

Rigidité. La flexibilité constitue un axe de la taxonomie de Céret [Céret *et al.* 2013]. Elle qualifie la façon dont une méthode doit être plus ou moins adaptée au contexte du projet et aux besoins [Céret *et al.* 2013]. Nous définissons la rigidité à l'inverse de la flexibilité. Nous pensons qu'il est nécessaire de fournir des processus rigides et formels, avec une identification des activités, des rôles pour les planifier et les réaliser, et des outils pour instrumenter et automatiser le processus. La rigidité est nécessaire dans un contexte de développement pluridisciplinaire pouvant impliquer de la propriété industrielle.

Itératif & Incrémental. Ces deux caractéristiques, identifiées dans le chapitre précédent comme fondamentales pour la plupart des modèles de processus, constituent la base d'un développement modulaire de tout système et sont applicables au développement des systèmes embarqués. Elles sont présentes dans la taxonomie de Céret [Céret *et al.* 2013]. Dans le contexte de l'ingénierie des systèmes embarqués, un *processus itératif* et un *développement incrémental* sont essentiels afin d'avoir la possibilité de mesurer *l'avancement du développement* et de réduire le *temps de développement* de ce dernier. Les différentes spécificités des systèmes embarqués, à savoir les systèmes distribués ou bien multi-fonctionnels font des systèmes embarqués des bons candidats pour les développements itératifs.

Traçabilité. La *traçabilité* joue un rôle essentiel pour le développement des systèmes embarqués et pour leur certification. Elle a besoin d'être gérée au niveau des processus, des langages et des outils. Broy [Broy 2006a] souligne l'importance de l'ingénierie des besoins comme base de conception des systèmes embarqués. Le respect de l'ingénierie des besoins, implique une

forte traçabilité tout le long du processus. La traçabilité est une caractéristiques des processus répondant aux spécificités de l'ingénierie des systèmes embarqués telles que la *gestion de la sous-traitance*, mais surtout permet *l'explicabilité et la certification des systèmes.*

Réutilisabilité. La *réutilisabilité* permet de réduire le *temps de développement.* Elle peut prendre différentes formes et intervenir à tous les niveaux du processus de développement : réutilisation de spécifications, tout comme réutilisation de bouts d'implémentation. Elle est nécessaire également pour amortir les coûts de production en préconisant la réutilisation d'une même plate-forme ou des mêmes composants d'une plate-forme pour toutes les gammes d'un produit d'une même série, tout en l'adaptant à la gamme en question.

Rôles & Responsabilités. Rôles & responsabilités sont introduits dans la taxonomie de Céret [Céret *et al.* 2013]. Dans cette taxonomie, les rôles sont divisés en deux catégories : les rôles internes et externes. L'explicitation des rôles et des responsabilités dans le processus de développement de systèmes embarqués nous semble très importante. Par exemple, dans [Gérard 2000], Gérard définit deux rôles dans le contexte de l'ingénierie automobile : le constructeur automobile et le fournisseur. On peut également distinguer pour ces deux rôles deux responsabilités distinctes : le premier *donne des ordres* (Gérard emploie la terminologie de *donneur d'ordre*) et le second les *exécute.* Dans [Broy 2006a], quelques rôles sont identifiés : les architectures logiciels et matériels, pour le développement du système embarqué en lui-même, et *l'orchestrateur* pour la synchronisation et la planification des développements. Sangiovanni-Vincentelli [Sangiovanni-Vincentelli 2007] identifie des rôles différents en fonction du domaine d'activité ciblé. Par exemple, les chaînes de production de communication mobiles nécessitent des développeurs applicatifs, des fournisseurs de service, des fabricants de produits, de semi-conducteurs et de composants. Dans le domaine de l'automobile, il identifie sensiblement les mêmes rôles que Gérard [Gérard 2000] tout en les explicitant. Par exemple, la terminologie de *fournisseur* de Gérard s'identifie aux fournisseurs tiers 1, 2 ainsi qu'aux producteurs.

Rôles et responsabilité permettent de répondre à un besoin *d'ingénierie interdisciplinaire, avec et une gestion de la sous-traitance et de la propriété intellectuelle.* De plus, le rôle de *chef de projet* permet de mesurer *l'avancement du développement.*

Parallélisme. Le parallélisme est une caractéristique essentielle des processus de développement de systèmes embarqués. Il permet de réaliser différentes étapes du processus en même temps, afin d'en réduire le temps de développement et le coût. Cette caractéristique apparait à deux dimensions pour les systèmes embarqués. D'une part, elle permet la collaboration d'équipes différentes, travaillant sur l'application et la plate-forme du système embarqué. D'autres part, le développement de systèmes embarqués faisant intervenir de grandes équipes de développement, il est nécessaire de pouvoir attribuer des activités aux différents constituants d'une équipe de développement, que ce soit de l'application ou de la plate-forme. Broy [Broy 2006a] souligne que le parallélisme nécessite un orchestrateur, c'est à dire une personne permettant d'organiser, de synchroniser et d'orchestrer les différentes équipes et activités réalisées.

Le parallélisme est une réponse des processus de développement pour satisfaire les besoins de l'ingénierie des systèmes embarqués. Elle cible le besoin de développer rapidement et ainsi réduire les coûts. Nous pensons que pour satisfaire cette spécificité, le processus de développement d'une méthode de systèmes embarqués *doit permettre un haut niveau de parallélisme et prendre en charge la synchronisation des développements parallèles.*

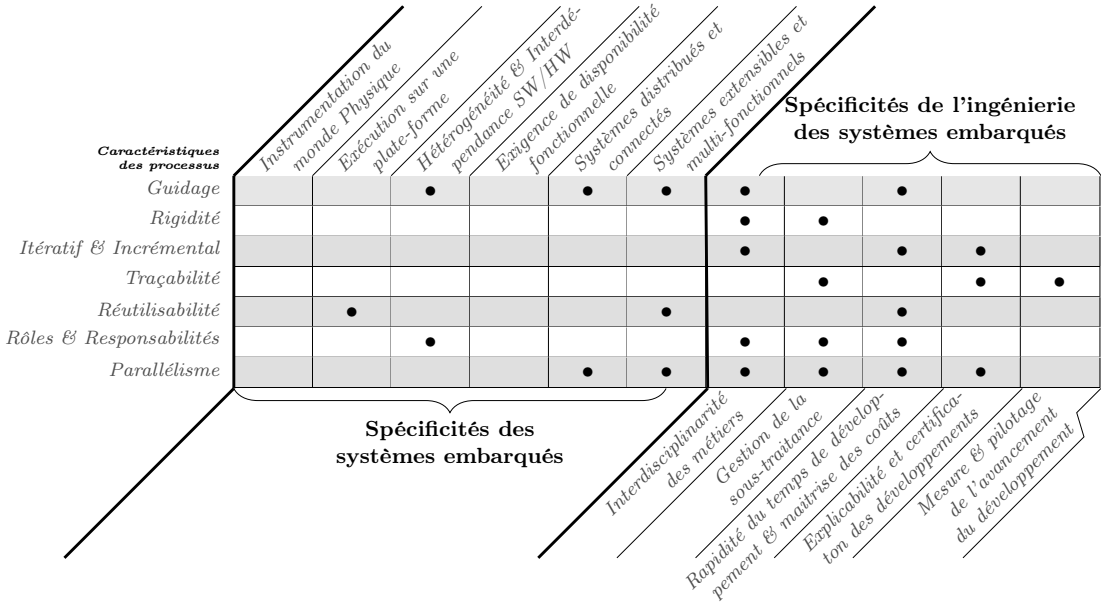


TABLE 4 – Résumé des caractéristiques des processus de développement de systèmes embarqués

Synthèse. Le tableau 4 synthétise les différentes caractéristiques identifiées et la manière dont elles répondent aux spécificités des systèmes embarqués et de leur ingénierie. Il compare les caractéristiques des processus verticalement aux spécificités des systèmes embarqués et de leur ingénierie verticalement. Le tableau est découpé verticalement en deux. La partie gauche relève des spécificités des systèmes embarqués. La partie droite quant à elle relate des spécificités de leur ingénierie. Un symbole (•) illustre une réponse de la caractéristique à la spécificité ciblées. Ainsi, par exemple, la caractéristique *réutilisabilité* répond à la spécificité *rapidité du temps de développement* du côté de l'ingénierie des systèmes embarqués et plus particulièrement pour les *systèmes extensibles et multi-fonctionnels*.

1.2 Les caractéristiques des langages

Afin de tenir compte des différentes spécificités des systèmes embarqués (systèmes distribués, extensibles, multi-fonctionnels, etc.) que nous avons identifiées dans le chapitre 2 le langage doit permettre de réduire la complexité de leurs développements. Nous les résumons dans cette partie autour de cinq caractéristiques : multi-vues, composabilité, abstraction, support à l'hétérogénéité et degré de formalisme.

Multi-vues. Les systèmes logiciels sont traditionnellement conçus autour de différentes vues représentant chacune une sous-partie du système. Les vues sont complémentaires et représentent le même système selon des points de vue différents. Elles apportent une simplification du problème du développement, chacune de ces vues ne laissant apparaitre que les informations utiles à un niveau de modélisation donné. Les vues sont traditionnellement définies selon trois dimensions : structurelle, fonctionnelle et dynamique. Par exemple, le langage UML [Object

Management Group 2010] propose les diagrammes de classes et de composants pour modéliser structurellement un système logiciel, le diagramme de cas d'utilisation pour représenter les fonctionnalités que le système à concevoir offre et enfin les diagrammes d'activités, de machine à états, etc. pour modéliser le comportement du système, c'est à dire de quelle manière ils implémentent les fonctionnalités.

En système embarqué, au même titre qu'en génie logiciel, modéliser ces trois aspects d'un système permet de structurer le développement, d'effectuer des analyses et simulations sur les modèles et à terme de générer du code fonctionnel (squelette du code à partir de vues structurelles, implémentation du fonctionnel à partir des vues dynamiques). À la différence des systèmes logiciels traditionnels, la conception de systèmes embarqués nécessite également de modéliser la plate-forme sur laquelle il s'exécute. Ainsi, la conception s'appuie sur des vues indépendantes et spécifiques à la plate-forme (on parle de *Platform Independent Model* et *Platform Specific Model* [Obj 2003]). Enfin, la complexité du développement des systèmes embarqués est accentuée par le nombre de métiers et de profils différents impliqués dans leur développement et qui ne partagent pas la même vision du système.

En cela, ces langages doivent permettre de gérer un ensemble de vues du même système. Ces vues permettent de visualiser le système selon l'aspect structurel, dynamique ou encore fonctionnel, mais aussi en considérant l'application indépendante de la plate-forme d'exécution d'une part, et spécifique à cette dernière d'autre part. L'ensemble de ces vues complémentaires donne la conception complète du système.

Composabilité. La composabilité au sein des systèmes embarqués est l'un des défis majeurs identifiés par [Sifakis 2011]. Commune à tous les types d'ingénierie, la construction par composition permet une conception individuelle des composants qui sont ensuite assemblés en un composant plus large. Des règles de composition assurent le bon assemblage des composants, notamment la préservation des propriétés d'un comportement et l'équivalence de son comportement. Les langages *modulaires* permettent d'exprimer la composition et de décomposer un problème en sous-problèmes plus simples à résoudre. Elle permet de concevoir automatiquement les spécifications d'un système basées sur celles de sous-systèmes, ou composants [Broy 2009]. Selon [Broy 2003], *pour gérer les larges systèmes, la composition devrait toujours être hiérarchique.*

La composabilité est d'autant plus importante au sein des systèmes embarqués que les composants à assembler sont souvent hétérogènes et potentiellement géo-localisés pour les systèmes distribués. En cela, le langage doit permettre une composition de composants hétérogènes selon des règles de composition formellement définies.

Support à l'abstraction. Les techniques d'*abstraction* et de *raffinement* sont discutées dans [Broy 2003, Henzinger & Sifakis 2007, Sifakis 2011, Gamatié *et al.* 2011]. Les travaux de [Broy 2003] donnent une vision formelle de ces concepts pour les systèmes embarqués. L'abstraction réduit la complexité de conception d'un système embarqué en cachant un certain nombre de détails à un niveau d'abstraction donné. À l'inverse, le raffinement permet, à un niveau d'abstraction, de faire apparaitre des détails qui ne sont pas présents dans des niveaux plus abstraits. Ces deux techniques sont liées à la notion de *niveau d'abstraction*. Ce dernier peut être changé par raffinement (on descend d'un niveau d'abstraction) ou abstraction (on monte d'un niveau). Un système peut alors être représenté par des modèles à un niveau d'abstraction donné et raffiné par raffinements successifs ou hiérarchiques [Henzinger & Sifakis 2007, Gamatié

et al. 2011]. Le terme *raffinement* s'applique à beaucoup de choses : modèle, mais aussi processus dans lequel les relations de raffinement définissent la démarche de développement [Broy 2003].

L'abstraction est un concept fondamental dans le cadre des systèmes embarqués. Sifakis et Henzinger [Henzinger & Sifakis 2007, Sifakis 2011] définissent l'abstraction hétérogène. Elle fait intervenir des *« styles » de modèles différents*. Par exemple, l'exemple donné par [Henzinger & Sifakis 2007] est celui de la porte logique fournissant une valeur booléenne abstrayant la valeur réelle délivrée par un transistor. À plus grande échelle, nous pouvons retrouver cette abstraction au niveau de l'application du système embarqué. Développée en premier abord indépendamment de toute plate-forme d'exécution, elle se situe à un niveau d'abstraction cachant les interactions avec l'environnement physique. Cette application est ensuite raffinée lors de son implémentation avec la plate-forme pour faire apparaitre les contraintes physiques liées à l'environnement d'exécution. Si Sifakis, Henzinger et Broy voient en l'abstraction une technique permettant de répondre aux spécificités des systèmes embarqués (interaction avec l'environnement physique, hétérogénéité et interdépendance entre le logiciel et le matériel), elle permet également la maitrise des coûts et des temps de développement [Gamatié *et al.* 2011, Keutzer *et al.* 2000]. En raisonnant sur les modèles abstraits, il est possible de réaliser des analyses et simulations sur le comportement attendu d'un système avant de le réaliser physiquement (un jeu de masques pour systèmes sur puce coutait cinq millions de dollars en 2000 [Keutzer *et al.* 2000]). Monter en abstraction favorise également une réutilisation efficace du développement [Gamatié *et al.* 2011] et contribue à diminuer l'ingénierie non récurrente [Keutzer *et al.* 2000].

Support à l'hétérogénéité. L'hétérogénéité se manifeste dans les systèmes embarqués de plusieurs façons : abstraction hétérogène telle que nous l'avons discutée dans le paragraphe précédent, mais aussi toutes les hétérogénéités discutées dans la section 1 : hétérogénéité logicielle et matérielle, hétérogénéité de distribution, d'interaction, temporelle.

Henzinger et Sifakis distinguent deux gros problèmes à l'hétérogénéité : du point de vue des systèmes embarqués, l'hétérogénéité complexifie la composition de sous-systèmes hétérogènes en plus gros [Henzinger & Sifakis 2007]. Ces derniers ne partagent pas forcément les mêmes sémantiques d'exécution et d'interaction. Du point de vue de leur ingénierie, la transformation et l'intégration des points de vue des différents profils impliqués dans la conception d'un système embarqué alourdit le processus de conception. Sangiovanni-Vincentelli partage cette vision en pointant l'hétérogénéité des composants comme étant à l'origine de la complexité d'intégration entre le logiciel et le matériel [Sangiovanni-Vincentelli 2007]. Sifakis [Sifakis 2011] insiste sur le besoin d'unifier les développements logiciels et matériels autour d'une seule sémantique :

> « Les descriptions du système utilisées tout au long des processus de développement devraient être basées sur une seule sémantique de modèles afin de maintenir une cohérence globale en garantissant qu'une description à une étape n+1 rencontrent les propriétés essentielles d'une description à l'étape n. La sémantique des modèles doit être suffisamment expressive pour inclure l'hétérogénéité des composants. » [Sifakis 2011]

Cet avis est partagé par différents auteurs [Broy 2006a, Poulhiès *et al.* 2006, Vanderperren *et al.* 2008, Assayad 2009]. Ainsi, le langage doit servir de support à l'hétérogénéité.

Degré de formalisme. Parmi les langages de développement de systèmes, Fraser et al. distinguent trois catégories [Fraser *et al.* 1994] : les langages formels ayant une syntaxe et une

sémantique rigoureusement définies, les langages semi-formelles définissant précisément une syntaxe mais dont la sémantique est imprécise, et enfin les langages informels ne définissant pas précisément la syntaxe et la sémantique [Dupuy-Chessa 2011]. Les langages formels permettent de concevoir des systèmes en garantissant la conception par des analyses et des simulations et de prouver qu'une conception est correcte par construction. Néanmoins, ce type de langages ne permet pas de valider le système conçu depuis les spécifications de ce dernier. De plus, un important manque de ces langages est leur compréhension et leur simplicité [Broy 2009].

À l'inverse, les langages de modélisation permettent de concevoir un système à un niveau d'abstraction favorisant une compréhension globale d'un système par toutes les personnes impliquées dans son développement. Ils permettent de modéliser tous les aspects d'un système, du fonctionnel jusqu'à l'implémentation. En ce sens, ils permettent de valider l'implémentation à partir des spécifications. Néanmoins, ils n'offrent pas le même niveau de formalisation permettant de réaliser des analyses prouvant la bonne construction d'un système.

Broy souligne l'écart qui se situe entre ces deux pratiques [Broy 2009]. Dans le cas des systèmes embarqués, les deux aspects sont importants. D'une part, afin d'assurer l'explicabilité et la certification des systèmes, le langage doit offrir un niveau de compréhension suffisant tout en garantissant sa bonne construction. D'autre part, les systèmes embarqués impliquent de gros développements dans lequel sont impliqués divers profils devant avoir une compréhension commune du système. Dans ce cas de figure, le langage doit être aussi simple et compréhensible que possible afin d'éviter tout ambiguïté de compréhension entre les différentes personnes.

Synthèse. Le tableau 5 synthétise les différentes caractéristiques identifiées que doivent posséder les langages pour répondre aux spécificités des systèmes embarqués et de leur ingénierie. Il compare les caractéristiques des langages aux spécificités des systèmes embarqués et de leur ingénierie. Tout comme pour le tableau 4, le tableau 5 est divisé en deux parties, la partie de gauche pour les spécificités des systèmes embarqués, celle de droite pour leur ingénierie.

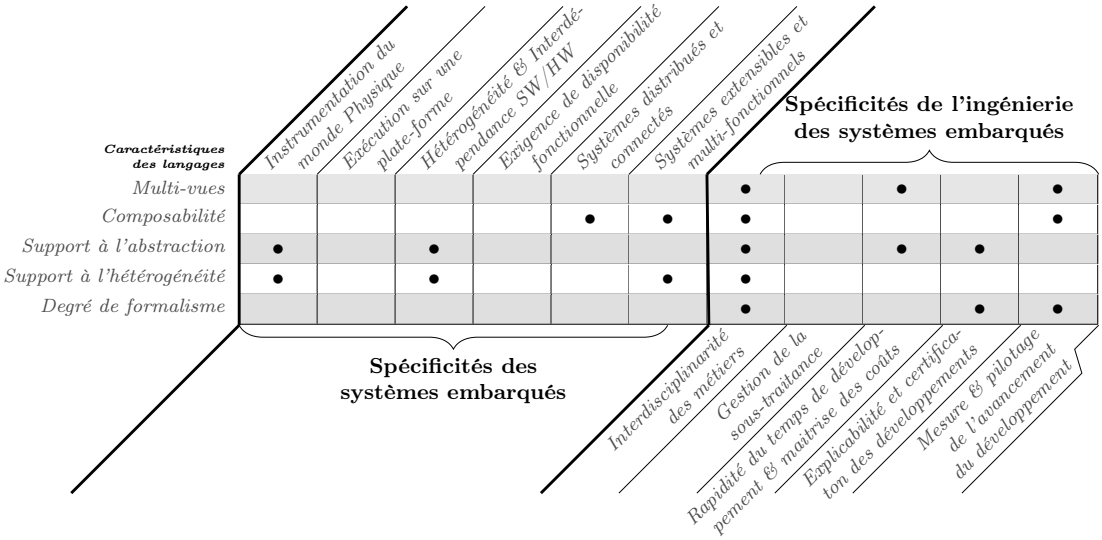


Table 5 – Résumé des caractéristiques des langages de développement de systèmes embarqués

1.3 Les caractéristiques des outils

Dans le chapitre précédent, nous évoquions l'outil comme l'une des composantes constituant les méthodes de développement. Le support par l'outillage est considéré comme fondamental dans cette discipline alliant la conception à des méthodes de vérifications formelles, allant de la simulation aux analyses statiques. Ce support de l'outillage est notamment discuté dans [Broy 2006a, Henzinger & Sifakis 2007, Biehl 2010].

Henzinger et Sifakis insistent sur la nécessité de posséder des outils de conception assistée par ordinateur (CAO) comprenant des outils d'analyse permettant la construction de systèmes robustes en assistant les ingénieurs dans le développement continu [Henzinger & Sifakis 2007]. Broy justifie le besoin d'outils support par la quantité de lignes de code écrites pour le développement de systèmes à prépondérance logicielle dans le secteur de l'automobile cité en exemple dans [Broy 2006a]. Il regrette le manque de chaîne d'outils assurant le développement de systèmes embarqués sur tout un processus de développement alors que l'ingénierie des systèmes embarqués possède selon lui d'excellents outils de développement mais de licences différentes et fragmentées ne couvrant pas tout le cycle de développement. Le manque de fondation commune de ces outils – pas de structure interne commune, niveaux de formalisme différents, particulièrement pour les outils de spécification – est un frein ne permettant pas l'intégration syntaxique et sémantique des outils [Broy 2006a].

L'intégration et l'interopérabilité des outils sont au cœur des travaux de Biehl [Biehl 2011, Biehl *et al.* 2012]. Dans [Biehl *et al.* 2012], les auteurs spécifient des chaînes d'outils et des générateurs pour réaliser l'intégration et les dépendances des divers outils composant la chaîne. Le langage de modélisation proposé sur lequel se repose cette intégration se nomme le « *Tool Integrated Language* » (TIL) [Biehl 2011]. Le coût estimé selon Törngren de la conception d'un modèle d'intégration entre des outils divers – COTS, outils non libre à licence commerciale, etc. – est de l'ordre de deux à quatre semaines [Törngren 2013b, Törngren 2013a].

Dans la plupart des travaux autour des systèmes embarqués, les auteurs se contentent de citer les outils comme support à un langage ou à un processus de développement, sans toutefois souligner l'importance d'un bon outillage et l'impact sur les-dits processus et langages. Nous proposons dans cette partie d'étudier les caractéristiques de l'outillage qui nous paraissent fondamentales. Néanmoins, d'un point de vue des méthodes, l'outillage ne répond pas directement aux spécificités des systèmes embarqués mais servent de support aux processus et langages qui y répondent. Aussi, nous ne terminerons pas cette partie sur un tableau synthétisant en quoi répondent les caractéristiques des outils pour les spécificités des systèmes embarqués et de leur ingénierie puisqu'ils y répondent aux travers des processus outillés et des langages supportés.

Support au processus. L'outil sert de support au processus [Broy 2006a, Henzinger & Sifakis 2007, Biehl 2010]. Il permet d'assurer le guidage dans ce dernier et d'éviter d'en dériver. En outre, le développement d'un système embarqué de façon itérative, incrémentale et parallèle doit s'appuyer sur un outillage afin d'assurer la cohérence globale des développements et la pérennité et la cohésion de la solution développée. L'aspect traçabilité du processus peut également être outillé afin de permettre de tracer le développement tout au long du processus.

Support au langage. L'outil doit permettre de supporter pleinement toutes les caractéristiques du langage de conception des systèmes embarqués. Notamment, la gestion de plusieurs vues (fonctionnelles, structurelles et dynamiques) dans un outil est fondamentale pour le dé-

veloppement de systèmes embarqués [Henzinger & Sifakis 2007, Törngren 2013b]. La composabilité de composants du systèmes peut par ailleurs être assistée par l'outillage qui assure la composition par des analyses et des vérifications de cohérence. L'outil peut en outre fournir des vues et des fonctionnalités selon le degré de formalisme ou le niveau d'abstraction souhaité.

Support à la gestion de projet. Comme nous l'avions évoqué dans la partie 1.1, le processus doit permettre de gérer plusieurs rôles et responsabilités. En outre, il doit permettre de découper le travail du chef de projet ainsi que ceux des développeurs. L'outil doit donc permettre un développement collaboratif ainsi qu'un suivi des développements effectués.

Le *support à la gestion de projet* est selon nous une caractéristique fondamentale des outils puisqu'elle permet de répondre favorablement aux spécificités de l'ingénierie des systèmes embarqués : maitrise de larges équipes, gestion de la sous-traitance et de la propriété industrielle, mesure de l'avancement du développement, maitrise des coûts, etc.

Support à la gestion de versions. La pluralité des équipes et le besoin de mesurer et piloter efficacement le développement nécessitent des outils servant de support à la gestion de versions. En particulier, le développement collaboratif nécessite des outils de gestion de versions. La rapidité du développement entraine un besoin de réutilisation d'un projet sur l'autre. De tels aspects pris en compte au sein d'un outil améliore l'efficacité du développement.

2 Étude comparative de différentes méthodes et approches

Dans cette section nous étudions différentes méthodes et approches pour la modélisation de systèmes embarqués : ACCORD/UML [Gérard 2000], une méthode pour la conception de systèmes embarqués temps-réels pour l'automobile, Metropolis [Balarin *et al.* 2002, Balarin *et al.* 2003, Sangiovanni-Vincentelli 2007], un environnement de conception intégrée de systèmes électroniques, BIP [Basu *et al.* 2011], un framework permettant la conception de systèmes basés sur la composition de composants hiérarchiques, et enfin MOPCOM [Aulagnier *et al.* 2009] une méthode permettant la conception de systèmes sur puce SoC [1]. Cette étude met en évidence la façon dont les méthodes actuelles prennent en compte les caractéristiques énoncées dans la section précédente. Nous étudions chacune des méthodes et notons la manière dont elle supporte les caractéristiques des processus, langages et outils.

○ (a) Non traité ◐ (b) Partiellement traité ● (c) Pleinement traité

FIGURE 17 – évaluateur de la prise en compte des caractéristiques

Les symboles de la figure 17 permettent d'évaluer et de qualifier la prise en compte des caractéristiques des processus, langages et outils par les méthodes de développement. Une pleine lune (○) signifie que la méthode ne prend pas en compte la caractéristique, une demi-lune (◐) signifie qu'elle le prend en compte partiellement ou imparfaitement et enfin une nouvelle lune (●) signifie que la caractéristique est pleinement traitée dans la méthode étudiée.

1. En anglais, System on Chip (SoC).

2.1 ACCORD/UML

ACCORD/UML [Gérard 2000] est une méthode de développement de systèmes embarqués pour le secteur automobile. Elle s'appuie sur les approches orientées modèles et s'articule autour d'un cycle de développement classique. Le langage utilisé est UML avec l'application de profils spécifiques pour le développement temps-réel de systèmes embarqués pour l'automobile.

Le processus. La vue d'ensemble du processus de la méthode ACCORD/UML est illustrée sur la figure 18. Le processus s'articule autour de trois phases : analyse préliminaire, analyse détaillée, et conception / réalisation. Il est défini comme itératif et continu [Gérard 2000]. Sur la figure 18, le guidage du processus est représenté par les flèches descendantes. Sur la gauche de la figure, des flèches ascendantes semblent définir un retour arrière possible afin d'assurer le caractère itératif du processus, mais l'auteur ne le mentionne pas [Gérard 2000]. Le processus es orienté activités. Les produits sont des entrées ou le résultat d'activités

La figure 19 présente plus en détail l'enchainement des activités dans l'approche ACCORD/UML. Elle s'apparente selon l'auteur à la branche gauche du cycle de développement en V. Bien que Gérard définit le processus de la méthode ACCORD/UML comme étant itératif, le terme n'est pas explicité et n'est pas associé à une grandeur temporelle ou une notion d'incrément sur le développement. Enfin, le processus n'exhibe aucun rôle, autre que celui de *développeur de système*. Le parallélisme n'est pas abordé.

Le langage. L'apport de la méthode ACCORD/UML réside essentiellement dans son langage, optimisé pour la conception d'application temps-réel pour les systèmes embarqués de l'automobile. La figure 20 illustre les interactions entre les différents modèles permettant la construction de systèmes embarqués temps-réel. Le modèle global est découpé en trois sous-modèles, séparant les trois composantes fondamentales pour la conception de systèmes, la structure, le

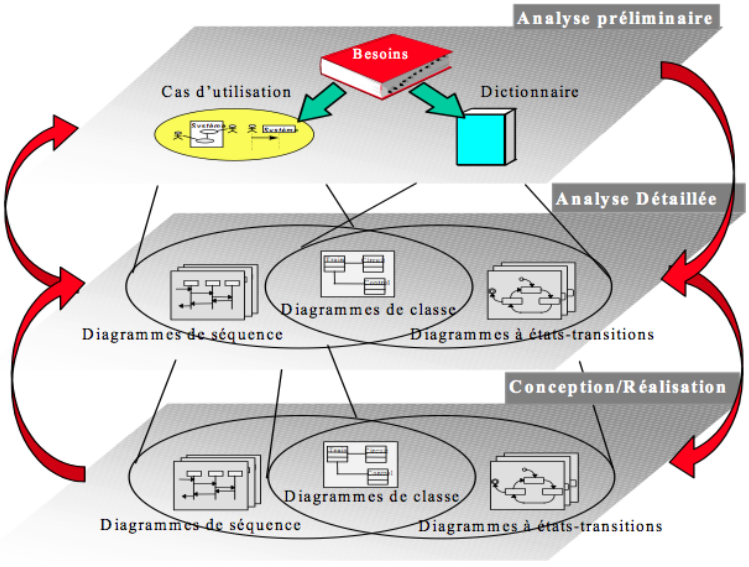


FIGURE 18 – Phases de l'approche ACCORD/UML [Gérard 2000]

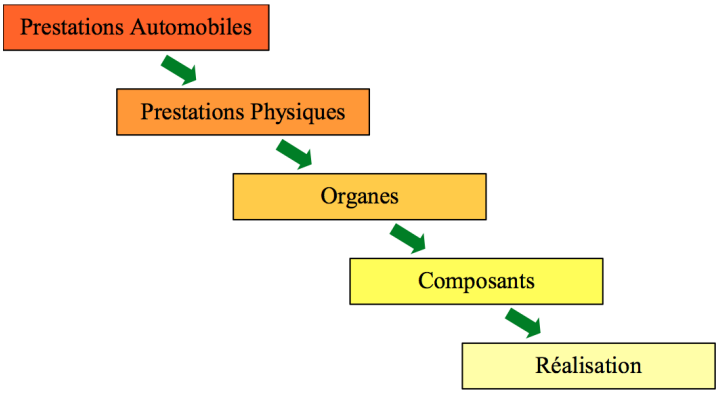


FIGURE 19 – Cycle de développement de ACCORD/UML [Gérard 2000]

comportement et les interactions. Le langage permet donc par ce découpage de gérer plusieurs vues complémentaires. En amont, le concept de *dictionnaire*, l'un des apports du langage dans ACCORD/UML, permet de capturer au niveau de la spécification les concepts clés du domaine ciblé, dans le but d'éviter les glissements sémantiques dus à l'utilisation d'un même terme par différentes personnes. Le dictionnaire permet d'éviter les confusions et les ambigüités de langages. Gérard suggère l'utilisation des dictionnaires dans un outil afin de tracer les exigences tout le long du processus [Gérard 2000].

Du point de vue structurel, ACCORD/UML s'appuie sur les diagrammes de classes d'UML. Le principal apport de la méthode dans cette partie est le concept d'Objet Temps-Réel (OTR). Un effort conséquent a été réalisé sur ce concept pour assurer l'aspect temps réel du langage.

Le comportement dans ACCORD/UML est défini dans le modèle comportemental par des machines à états UML [Object Management Group 2010]. ACCORD/UML propose une version restreinte de ces machines à états en y ajoutant deux points supplémentaires : le point de vue *protocole* spécifiant le comportement global de l'objet et le point de vue *déclenchement* permettant de spécifier des comportements particuliers tels que la réception de signaux [Gérard 2000]. Dans ACCORD/UML, chaque objet possède une machine à états.

La composition apparait à plusieurs niveaux dans le langage de la méthode de ACCORD/UML.

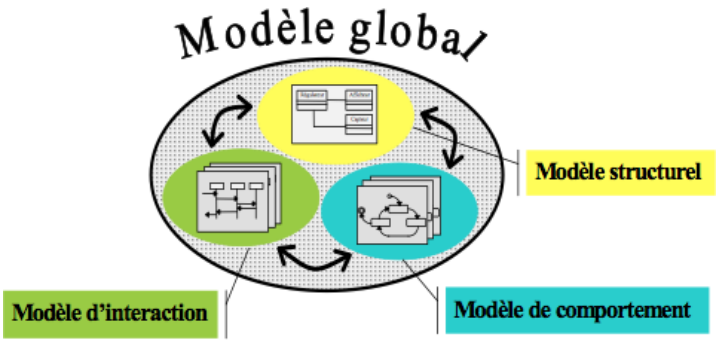


FIGURE 20 – Langage ACCORD/UML [Gérard 2000]

D'une part, le caractère intrinsèque des classes et de leur composition dans un modèle de classes, la composition des machines à états d'un objet selon les points de vue protocole et déclenchement, et enfin, la composition de plusieurs objets du point de vue comportemental, qui se traduit par l'émission et la réception de signaux par une machine à états.

Le langage supporte l'abstraction au travers du langage UML. Par exemple, le modèle de communication permet de spécifier différents types de message (synchrone, asynchrone, avec ou sans valeur de retour) par des méthodes UML. Cependant, il ne supporte pas l'hétérogénéité au sens où les objets définis sont purement logiciels et la plate-forme n'est pas modélisée.

Le support de l'outillage. La modélisation UML s'effectue avec l'outil *Objecteering*. Il est utilisé d'une part pour la modélisation dans ACCORD/UML des modèles issus de l'analyse préliminaire, détaillée ainsi de la phase de conception de l'approche, et d'autre part pour la génération de code et de fichiers exécutables. L'outil supporte les différentes activités du processus de la méthode ACCORD/UML mais Gérard ne précise pas si l'outil assure le guidage du processus, autrement dit, si un environnement dédié à la méthode et au suivi de l'exécution du processus a été personnalisé au sein d'Objecteering. L'outil permettant la gestion séparée des trois points de vue des modèles ACCORD/UML, nous pouvons dire que l'outil est multi-vues, sans pour autant être multi-utilisateurs, le processus n'exhibant aucun rôle ni responsabilité. Enfin, Objecteering n'offre aucun service de gestion de version ou de gestion de projet.

(a) Processus

Guidage	◐
Rigidité	○
Itératif & Incrémental	◐
Traçabilité	○
Réutilisabilité	○
Rôles & Responsabilités	○
Parallélisme	○

(b) Language

Multi-vues	●
Composabilité	◐
Support à l'abstraction	◐
Support à l'hétérogénéité	○
Degré de formalisme	◐

Support au processus	◐
Support au langage	◐
Support à la gestion de projet	○
Support à la gestion de versions	○

(c) Outils

TABLE 6 – Caractéristiques supportées par la méthode Accord / UML

Synthèse. Le tableau 6 résume les différentes caractéristiques prises en compte par ACCORD/UML. Côté processus, le guidage est assuré par un enchainement d'activités dans lequel les produits produits durant une activité deviennent les points d'entrée d'activités aval. Néanmoins, le processus n'explicite pas les états des produits, permettant d'évaluer si le passage à une phase aval est possible. L'aspect itératif est annoncé par l'auteur mais n'est pas explicité. ACCORD/UML s'appuie sur le langage UML et apporte des concepts orientés temps-réel. La

composabilité est assurée d'un point de vue structurel (diagramme de classes) et comportemental (diagramme de machines à états et émission / réception de signaux). La méthode ne gère que l'aspect applicatif d'un système ACCORD/UML et ne gère donc pas l'hétérogénéité. Enfin, le langage propose trois vues de modélisation complémentaires, les vues structurelle, comportementale et interactionnelle. L'outil utilisé est générique. Il permet de modéliser des systèmes avec le langage UML. Ainsi, l'outil supporte le langage proposé par Gérard, mais n'est cependant pas adapté pour coupler une traçabilité et une gestion de projet aux modèles produits, et n'est pas dédié à l'application du processus proposé dans ACCORD/UML.

2.2 Metropolis et l'approche orientée plates-formes

Metropolis [Balarin *et al.* 2002, Balarin *et al.* 2003, Sangiovanni-Vincentelli 2007] est l'implémentation la plus populaire des principes de l'approche orientée plates-formes [2] [Ferrari & Sangiovanni-Vincentelli 1999, Keutzer *et al.* 2000, Sangiovanni-Vincentelli *et al.* 2004]. L'approche orientée plates-formes est une évolution de l'approche orientée composants [Heineman & Councill 2001, Szyperski 2002, Krakowiak 2009]. Elle est motivée pour répondre aux principales difficultés rencontrées par les industries : horizontalisation de l'industrie, pression pour réduire le temps de mise sur le marché et enfin le coût de l'ingénierie non-récurrente. Elle consiste à partir du plus haut niveau d'abstraction dans lequel tous les détails d'implémentation sont cachés et à descendre dans la conception vers une implémentation finale des spécifications initiales en utilisant des instances de plates-formes à tous les niveaux d'abstraction. Dans ce contexte, une plate-forme est définie « *comme une librairie de composants pouvant être assemblés pour générer une conception à un niveau d'abstraction donné.* » [Sangiovanni-Vincentelli 2007]. Une instance de plate-forme est définie comme étant « un ensemble de composants qui sont sélectionnés d'une librairie (la plate-forme) et dont les paramètres sont définis. » [Sangiovanni-Vincentelli 2007].

Metropolis est défini comme étant un environnement intégré de conception de systèmes électroniques [Balarin *et al.* 2003]. Le projet est né du besoin d'unifier les outils composant les chaînes d'outils pour le développement de systèmes électroniques. Les auteurs argumentent que des outils non liés créent des incompréhensions parmi les différents parties prenantes du développement d'un système. Les transformations entre les outils, soient manuelles, soient traversant différents formats intermédiaires ne peuvent être utilisées en confiance avec une faible connaissance de l'ensemble du domaine.

Le processus. Metropolis se présente comme un framework unifié, présenté dans la figure 21. Le processus s'appuie sur les principes des approches orientées plates-formes pour supporter les activités classiques de modélisation. Ces activités classiques de conception de systèmes sont par exemple la spécification, l'architecture ou l'assemblage d'applications et de plates-formes. Il ne propose cependant aucun processus de développement permettant de guider et d'assister le développeur.

Le processus de l'approche orientée plate-forme est la combinaison d'une approche descendante et ascendante. L'approche descendante consiste à lier les fonctionnalités d'une conception vers une instance de plate-forme. L'approche ascendante consiste à construire une plate-forme en choisissant des composants pour la caractériser. Le processus est récursif et une instance de plate-forme apparait à chaque récursion. Le processus se termine lorsque tous les composants

2. En anglais, Component-based design (CBD).

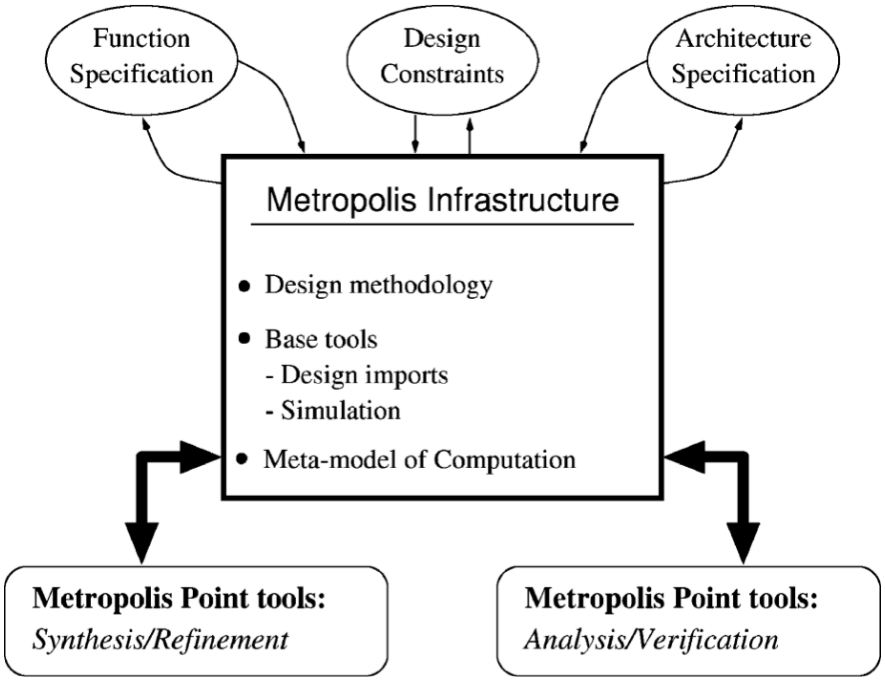


FIGURE 21 – Framework Metropolis [Balarin *et al.* 2003, Sangiovanni-Vincentelli 2007]

du niveau abstrait sont implémentés. Le nombre de récursions pour l'implémentation de tous les composants peut varier et ainsi les composants peuvent atteindre leur implémentation finale avant la fin du processus de récursion. Ce processus permet la construction de plates-formes par itérations successives et offre un haut niveau de support à l'abstraction. L'implémentation de fonctionnalités d'une instance de plates-formes est toujours considérée sur l'instance basse, menant à une forte indépendance entre la conception et l'implémentation.

Aussi, nous pouvons dégager les principales caractéristiques de l'approche orientée plates-formes. L'aspect incrémental par la construction successive de plates-formes basées sur des plates-formes plus abstraites, et l'association des fonctionnalités de l'application sur celles-ci ; la traçabilité assurée par l'association des fonctionnalités en fonction de l'instance de l'application conçue par rapport à l'application amont, ainsi que les ressources d'une plate-forme par rapport à la plate-forme aval.

Le langage. Le mécanisme interne de Metropolis est représenté par un méta-modèle, appelé le *Méta-modèle Metropolis* [The Metropolis Project Team 2004]. Il embarque un ensemble de générateurs afin d'importer de nombreux modèles de programmation et langages formels. Le méta-modèle supporte des concepts hétérogènes issus du matériel comme du logiciel.

Balarin [Balarin *et al.* 2003] spécifie trois points de vue pour la conception de systèmes dans le langage Metropolis : fonctionnel, architectural et le mapping de l'un sur l'autre. En cela, le langage proposé est multi-vues. De plus, dans l'esprit des approches orientées objets, ce langage propose un mécanisme de récursion de plates-formes [Balarin *et al.* 2003]. Ce mécanisme offre une perspective modulaire du langage, basée sur la récursion au niveau des plates-formes.

Du point de vue de la composabilité, le langage propose un mécanisme basé sur le concept d'interfaces. Par exemple, dans un modèle fonctionnel, un *medium* est un concept du méta-modèle permettant d'interfacer les différents processus entre eux.

Le support de l'outillage. L'outillage dans Metropolis est discuté dans [Balarin *et al.* 2003] ainsi que dans [Sangiovanni-Vincentelli 2007]. Dans [Sangiovanni-Vincentelli 2007], Sangiovanni-Vincentelli alerte sur le fait que l'outil Metropolis n'a pas pour vocation à couvrir toutes les activités de conception de systèmes, mais propose un mécanisme pour préserver toutes les informations de conception d'un système dans le langage. Libre à l'utilisateur d'interfacer l'outillage de Metropolis avec d'autres outils existants pour supporter les différentes activités d'ingénierie. Dans [Balarin *et al.* 2003], quelques fonctionnalités de l'outil, telles que la vérification de propriétés formelles, des écrans de simulation et des outils d'ordonnancement sont décrites. Ainsi nous pouvons décrire l'outil comme un support de conception permettant de visualiser un système en cours de conception sous les différentes vues offertes par le langage. Les autres caractéristiques que nous avions identifiées à propos de l'outillage ne sont pas prises en compte dans l'outil Metropolis.

Synthèse. Le tableau 8 résume différentes caractéristiques supportées par Metropolis. Le processus est basé sur les approches orientées plates-formes et à ce titre propose une structure itérative et incrémentale avec une traçabilité entre les différentes instances de plates-formes baties. La réutilisibilité se situe au niveau des composants dans une approche ascendante. Le langage utilisé est dédié, représenté par un méta-modèle. Il permet une composabilité à partir d'interfaces et permet de modéliser trois vues complémentaires. Si l'outillage supporte le langage, il ne couvre en revanche pas tout le processus de conception de systèmes embarqués et ne permet pas d'effectuer de la gestion de projets ou d'améliorer l'efficacité de la conception.

(a) Processus

Guidage	○
Rigidité	○
Itératif & Incrémental	◐
Traçabilité	◐
Réutilisabilité	◐
Rôles & Responsabilités	○
Parallélisme	○

(b) Language

Multi-vues	◐
Composabilité	◐
Support à l'abstraction	○
Support à l'hétérogénéité	◐
Degré de formalisme	◐

(c) Outils

Support au processus	◐
Support au langage	●
Support à la gestion de projet	○
Support à la gestion de versions	○

TABLE 8 – Caractéristiques supportées par la méthode Metropolis

2.3 Behavior, Interaction, Priority (BIP)

Behavior, Interaction, Priority (BIP) [Basu *et al.* 2011] est un framework développé au sein du laboratoire Verimag [Verimag Laboratory 2013a]. Il s'agit d'un framework pour la conception de systèmes basés sur les approches orientées modèles et composants. Il permet la construction de modèles composites et hiérarchiques dans lesquels chaque composant atomique est considéré en termes de comportement et d'interaction avec d'autres composants. L'aspect formel des modèles manipulés permet de garantir la preuve par construction de modèles de composants hiérarchiques basés sur des composants atomiques dont le comportement est clairement défini.

Le processus. Le flot de conception dans BIP est un flot orienté activités. Il est illustré dans la figure 22. Cette figure illustre les six activités du flot BIP par des rectangles verts, impliquant des produits en entrée et des produits conçus illustrés par des rectangles aux bords arrondis.

Les quatre principales activités de BIP sont : transformation d'un modèle d'application en un modèle d'application dans le langage BIP ; intégration de contraintes architecturales issues de la plate-forme matérielle d'exécution ; intégration du protocole de communication ; et enfin génération d'un code déployable. Deux activités additionnelles, de prévention de deadlock (*D-finder* [Bensalem *et al.* 2011]) et d'analyses de performances donne au processus son aspect itératif et incrémental. Cet aspect est illustré dans la figure par des flèches en pointillé (- - -▸) alors que le flot continu de conception est représenté par des flèches en trait plein (——▸).

Le processus est itératif, formalisé, rigide et guidé. Notamment, il s'appuie sur des vérifications et des analyses pour vérifier la cohérence des modèles produits, et dans le cas contraire, permet de les modifier. En revanche, l'aspect parallélisme des activités n'est pas traité, ni la traçabilité. Aucun rôle ou responsabilité des acteurs n'est défini. Le processus est essentiellement centré sur la construction par composition et suppose un partitionnement initial de l'application et la définition du déploiement des différents composants logiciels sur les différents processeurs.

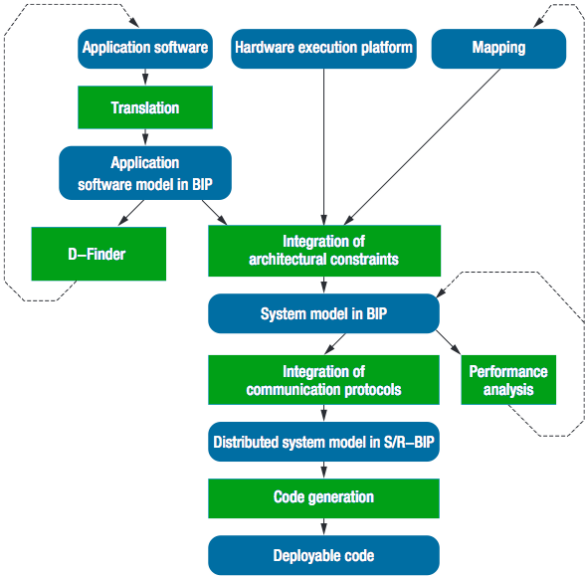


FIGURE 22 – Le flot de conception dans BIP [Basu *et al.* 2011]

Le langage. BIP se distingue des autres modèles à composants par son mécanisme d'exécution et de priorité des composants, basé sur la définition de politiques d'ordonnancement. Les connecteurs de BIP ont la particularité de ne posséder aucun comportement, les connecteurs et les priorités définis dans BIP étant indépendants des comportements. Le comportement des composants atomiques repose sur le formalisme des automates à états finis ou une extension des réseaux de Pétri incluant données et ports [Basu *et al.* 2011].

L'exemple de la figure 23 illustre la composition de deux composants atomiques pour le contrôleur du robot Dala [Verimag Laboratory 2013b]. Les deux composants, *Activity* et *Service Controller* sont composés dans un composite *Service*. Le contrôle est assuré par le *contrôleur de service*, les calculs par le composant *Activity*. Les comportements de chaque composant atomique sont représentés par des automates, les transitions étant définies par une garde – condition booléenne – ainsi qu'une action – une fonction définie en C/C++ [Basu *et al.* 2011]. Les connecteurs entre les composants sont représentés par le symbole ●—●, ceux entre un composant et son composite par le symbole ●- -●. Au bout de la connexion se trouve les noms des ports des composants interagissant. Non représenté sur la figure 23, les connections sont définies formellement à partir de garde et de fonctions de transfert. La connexion peut évoluer sous deux formes : *synchrone* ou *diffusion* (*broadcast*). Une priorité (non représentée sur la figure) permet de définir quelle est la transition qui sera franchie lorsque plusieurs sont actives en même temps. Le *Service proxy*, un autre composant ajouté permet d'illustrer l'encapsulation fournie par l'usage de ports et d'interfaces. Un déclencheur, représenté par le symbole ▶—●, permet de représenter l'action d'un trigger de la part d'un composant vis-à-vis d'un autre.

Les auteurs de BIP ont défini une théorie de compositions des composants BIP permettant de garantir la preuve par construction. Le langage BIP est considéré comme un langage hôte, c'est-à-dire que des langages sources (C, Lustre) peuvent être transformés en BIP et ainsi bénéficier de tout son outillage formel.

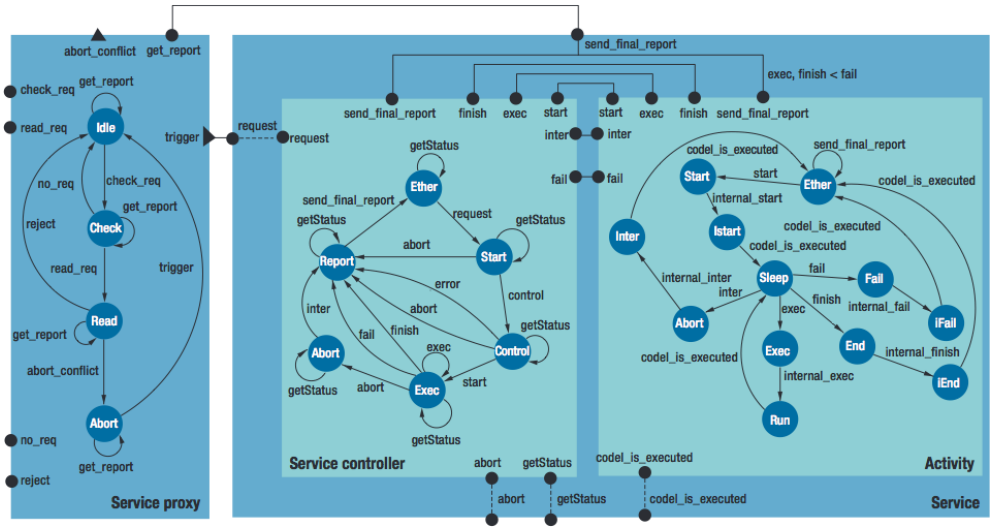


FIGURE 23 – Un example de composition en BIP [Basu *et al.* 2011]

Le support de l'outillage. L'outillage proposé comme support au processus et au langage BIP est illustré dans la figure 24. Il inclut des outils permettant de transformer des langages sources tels que du C, du DOL, etc. en un modèle BIP. Ces modèles sont utilisés tout au long des différentes activités du processus. L'outil supporte notamment les modèles BIP de l'application, et les modèles BIP de l'application distribuées sur l'architecture (*State/Relation-BIP*) incluant les protocoles de communication. Des outils de vérification et de validation, tels que le *D-Finder* permettent de vérifier la cohérence des modèles. Enfin, des outils de génération de code permettent de générer le code déployable sur des plates-formes cibles.

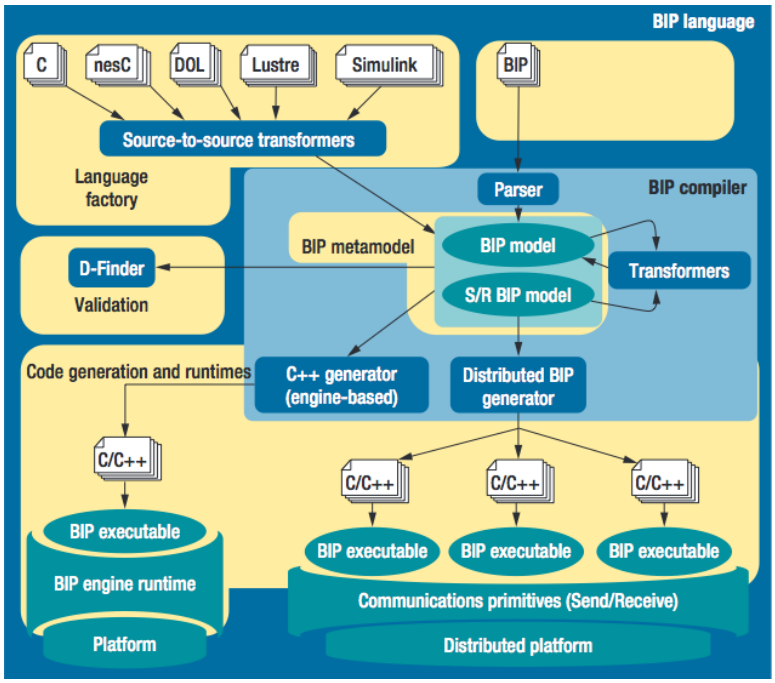


FIGURE 24 – Le flot de conception dans BIP [Basu *et al.* 2011]

Synthèse. Le tableau 10 synthétise les caractéristiques supportées par BIP. Le processus est parfaitement formalisé et guidé et définit les itérations et les incréments possibles dans le flot. Néanmoins, il n'aborde pas le travail collaboratif ou la gestion de projet.

La gestion de plusieurs vues semblent être partiellement traitée dans le langage, avec une séparation entre les aspects comportement et de communication. Le langage est basé sur une théorie formelle assurant la composition des composants BIP et des ports servant d'interface entre les différents composants.

L'outil est bien utilisé pour supporter toutes les phases du processus, en cela, nous pouvons dire qu'il supporte efficacement le processus dans sa globalité. Le fait de posséder un unique langage permet de garantir l'interopérabilité de tous les outils dédiés constituant la chaîne d'outils de BIP. Néanmoins, rien n'est dit sur les éventuels mécanismes de gestion de versions et de réutilisation intégrés, ni sur l'aspect multi-vues que pourrait posséder l'outil, les exemples illustrés présentent des modèles regroupant les différents aspects des composants.

(a) Processus

Guidage	●
Rigidité	●
Itératif & Incrémental	◐
Traçabilité	○
Réutilisabilité	●
Rôles & Responsabilités	○
Parallélisme	○

(b) Language

Multi-vues	◐
Composabilité	●
Support à l'abstraction	◐
Support à l'hétérogénéité	○
Degré de formalisme	●

(c) Outils

Support au processus	●
Support au langage	●
Support à la gestion de projet	○
Support à la gestion de versions	○

Table 10 – Caractéristiques supportées par la méthode BIP

2.4 MOPCOM

MOPCOM [Koudri *et al.* 2008, Aulagnier *et al.* 2009] est une méthode de co-développement pour la conception de systèmes sur puce et systèmes sur puce programmable basée sur des FPGA (Field Programmable Gate Array) [Koudri *et al.* 2008, Aulagnier *et al.* 2009]. Elle est basée sur les approches orientées modèles et reprend les modèles de l'approche Model-Driven Architecture (MDA), notamment sur la séparation entre les modèles indépendants des plates-formes d'exécution PIM [3] et les modèles spécifiques PSM [4]. La méthode s'appuie sur les langages MARTE [Object Management Group 2011] et SysML [Object Management Group 2012].

a) System Modeling Language (SysML)

SysML est une extension d'UML réutilisant une partie des modèles UML (machines à états, séquences, etc.) et ajoutant un ensemble de nouveaux concepts (par exemple le concept de blocs ou le mécanisme d'allocation) qui adressent les besoins de l'ingénierie des systèmes sur toutes les phases de conception (spécification, analyse, conception, vérification et validation) [Object Management Group 2012]. Les concepts supplémentaires de SysML sont implémentés sous la forme d'un profil UML. Le concept fondamental ajouté est celui du *bloc*, un concept modulaire héritant des propriétés des classes et des structures composites UML. Par rapport à UML, SysML apporte le *mécanisme d'allocation* et le *diagramme de spécification du besoin*. Le mécanisme d'allocation permet de naviguer dans un modèle en établissant des relations entre les éléments du modèle. Le diagramme de spécification permet de capturer les besoins. Un besoin spécifie une capacité ou une condition qui doit être satisfaite. Ce concept adresse la formalisation efficace des spécifications d'un systèmes avec diverses relations entre ces spécifications.

3. En anglais, Platform Independent Model (PIM).
4. En anglais, Platform Specific Model (PSM).

b) Modeling and Analysis of Real-Time Embedded Systems (MARTE)

MARTE [Object Management Group 2011] est un langage standardisé par l'Object Management Group (OMG) dans le but de répondre spécifiquement aux besoins des systèmes embarqués temps-réel. Il est proposé sous la forme d'un profil UML. Il cible deux activités fondamentales, la modélisation et l'analyse des modèles. Du point de vue de modélisation propre, MARTE apporte les concepts nécessaires pour modéliser les aspects embarqués et temps-réel des systèmes embarqués. Du point de vue de l'analyse, MARTE offre des notions afin d'annoter des modèles pour appliquer des techniques d'analyse. Le langage MARTE offre quatre paquetages principaux. Le premier est nommé « *MARTE foundations* » et définit les concepts de base utilisés dans les domaines de l'embarqué et du temps-réel, aspect temporel, gestion des allocations des ressources, etc. [5]. On peut y trouver le paquetage GRM (pour *Generic Resource Modeling*) pour modéliser des plates-formes (logicielles ou matérielles) de manière abstraite. Un second paquetage se nomme GCM (pour *General Component Model*). Ce dernier contient les concepts relatifs aux composants et à leur composition.

Le processus. La figure 25 présente une vue d'ensemble de la méthode MOPCOM. Le flot de conception est représenté dans un rectangle à trait discontinu. Les modèles fonctionnels sont représentés par des rectangle blancs à trait plein. Les modèles de la plate-forme sont quant à eux représentés par des rectangles gris à trait plein. Entre chaque modèle fonctionnel et de la plate-forme, une activité d'allocation est requise. Enfin, le flot de conception suggère des activités de génération de code pour générer du SystemC à des fins de simulation. Une fois la plate-forme d'exécution définie, des générateurs permettent également de générer du code C embarqué pour la cible logicielle et VHDL pour la cible matérielle.

La figure 26 détaille les trois niveaux de modélisation du processus de la méthode MOPCOM. Les trois niveaux se nomment successivement *niveau de modélisation abstrait*, *niveau de modélisation exécutable* et enfin *niveau de modélisation détaillée*. Le premier niveau permet une modélisation abstraite du fonctionnel et de la plate-forme. La plate-forme fait référence à une plate-forme d'exécution abstraite ou virtuelle dans laquelle les modèles de programmation et de communication sont explicités. L'approche utilise les résultats du MDA pour combiner ces deux modèles dans un modèle d'allocation. Le second niveau fait intervenir une première topologie d'une plate-forme physique composée de processeur, de mémoire, etc. Le modèle obtenu au niveau précédent est alors combiné avec cette nouvelle plate-forme. Il en résulte un second modèle d'allocation. Ce modèle est utilisé dans un but d'analyse d'ordonnancement. Le dernier niveau est sensiblement équivalent au précédent. Dans ce niveau, le modèle de la plate-forme est raffiné et permet à nouveau d'obtenir un modèle d'allocation permettant par la suite la génération du code, à la fois pour le logiciel et pour le matériel.

Nous pouvons assimiler ce processus au modèle en Y proposé par Capretz [Luiz Fernando Capretz 2005]. Plus précisément, le processus de la méthode MOPCOM est construit sur l'exécution de trois instances du modèle. Ainsi, l'approche permet une parallélisation des activités et un développement concurrent. Néanmoins, le processus ne définit aucun degré de parallélisme. Les activités sont guidées et assistées par des générations de code et de documentation, ainsi que des analyses et simulation. Il ne semble donc pas possible d'entamer une activité si les analyses ne vérifient pas le bon fonctionnement du développement. En cela, le processus

5. La spécification de MARTE remplace l'ancien profil *Schedulability, Performance and Time Profile* (SPTP) standardisé par l'OMG en 2005 [Object Management Group 2005]

proposé semble plus rigide que le processus proposé dans la méthode ACCORD/UML par Gérard. Dans [Koudri *et al.* 2008, Aulagnier *et al.* 2009], les auteurs n'exhibent aucun rôle ou responsabilité particulière utilisant le processus. L'aspect traçabilité semble traité mais uniquement sur l'aspect langage, la traçabilité dans le langage étant assuré par l'usage du stéréotype « Allocated » sur les blocs du modèle d'allocation [Aulagnier *et al.* 2009].

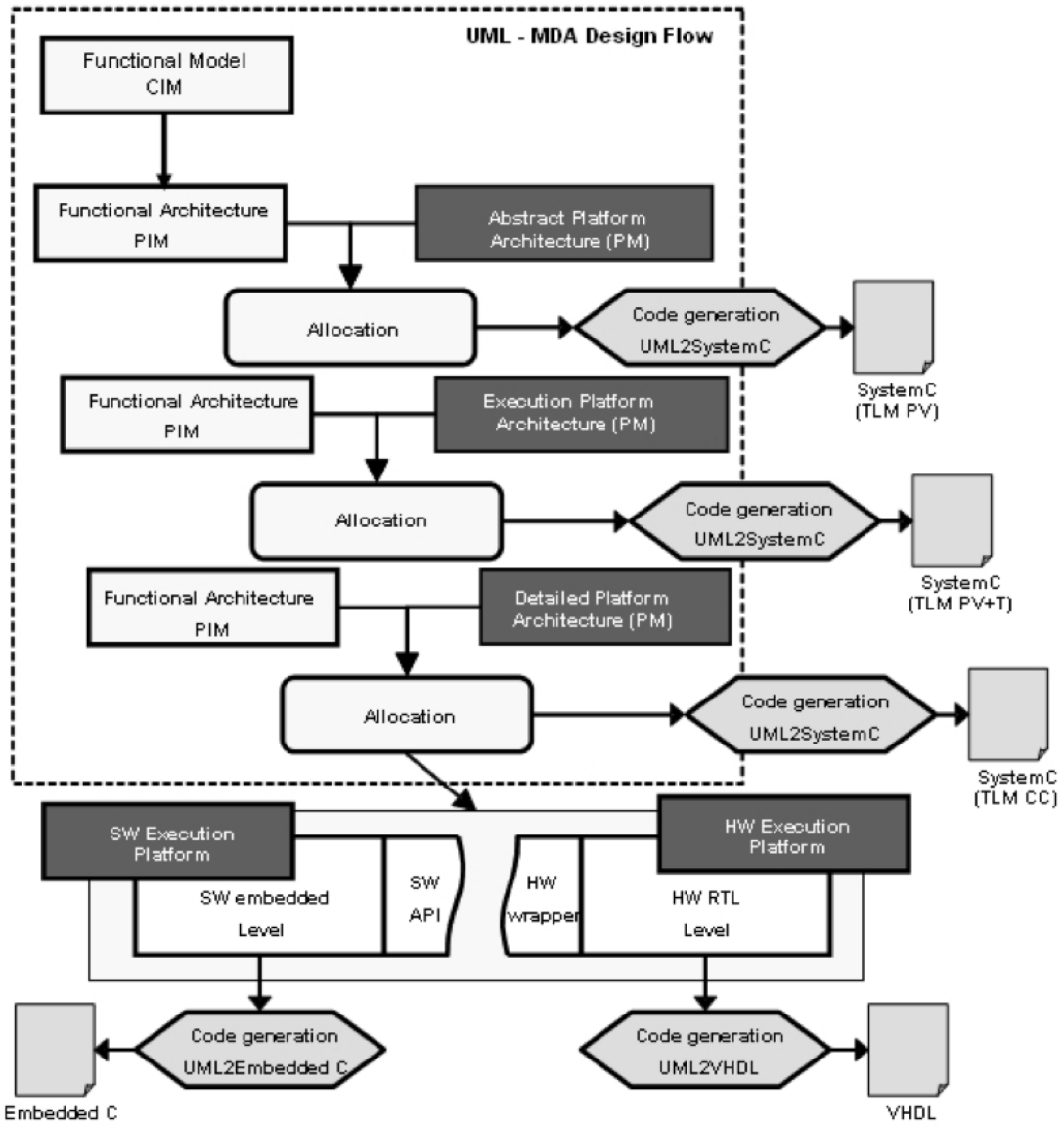


FIGURE 25 – Vue d'ensemble de la méthode de co-développement MOPCOM [Koudri *et al.* 2008, Aulagnier *et al.* 2009]

Le langage. Les deux langages utilisés dans le cas de la méthode MOPCOM sont SysML et MARTE. L'hétérogénéité de MARTE est utilisée pour spécifier à la fois les aspects matériels et logiciels d'un système embarqué. La définition de la plate-forme, à tous niveaux de modélisation confondus, s'appuie sur les diagrammes UML de composants [Object Management Group 2010], en application du profil de MARTE pour les aspects liés aux systèmes embarqués.

La traçabilité s'appuie sur le mécanisme de SysML [Aulagnier *et al.* 2009]. Ce mécanisme offre deux stéréotypes, ou concepts. Le stéréotype « allocate » permet d'associer deux éléments de différents types. Cette association est abstraite et laisse suggérer qu'une implémentation de cette allocation est définie lors de phases de conception plus concrètes. Le second stéréotype « allocated » s'applique à tout objet impliqué dans une relation d'allocation.

L'abstraction est gérée par les niveaux de plates-formes sur lesquels l'application est successivement implémentée, mais est néanmoins limitée au nombre figé de ces niveaux. De plus, rien n'est dit sur la façon d'implémenter l'aspect comportemental de l'application sur ces plates-formes. Enfin, l'aspect composabilité ne semble être traité que du point de vue structurel et non du point de vue comportemental. Rien ne semble présumer de l'aspect modulaire du langage, ni l'utilisation de plusieurs vues.

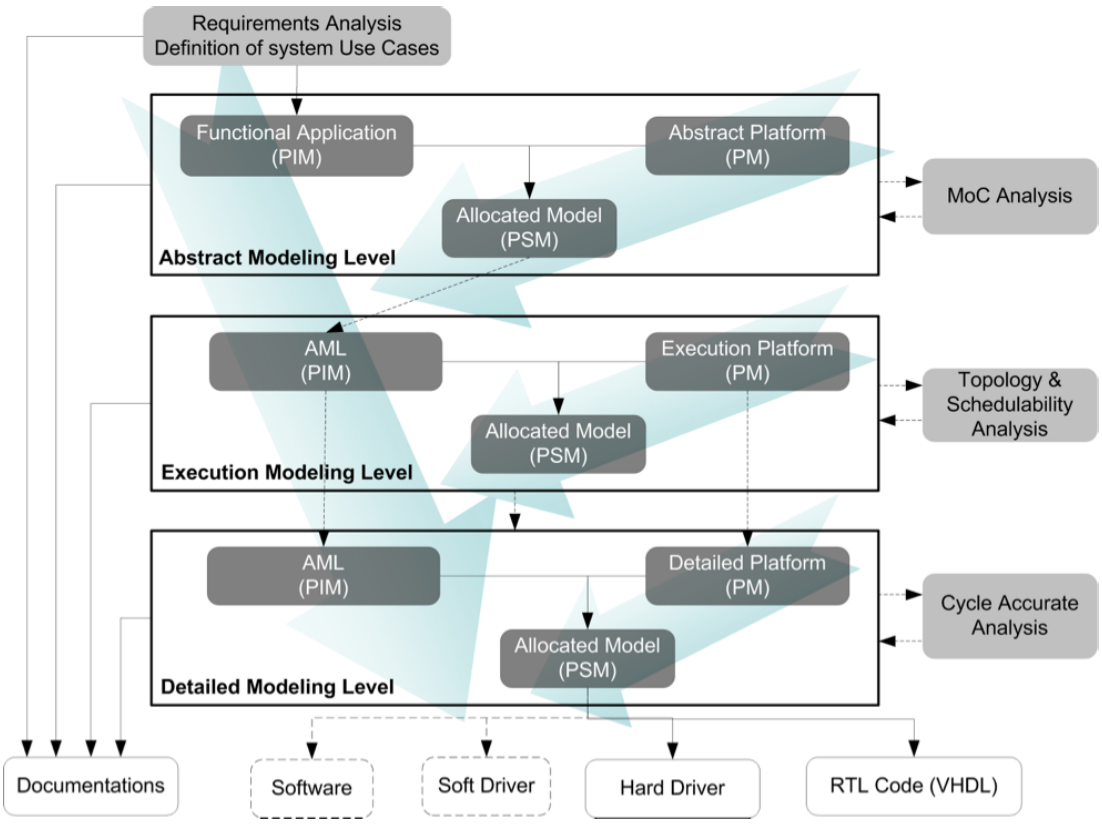


FIGURE 26 – Les niveaux d'abstraction de MOPCOM [Koudri *et al.* 2008, Aulagnier *et al.* 2009]

Le support de l'outillage. Le support de l'outillage pour la méthode MOPCOM est illustré par la figure 27. La chaîne est constituée d'outils sous licence libre ou commerciale permettant la conception, l'édition de modèles et la génération de code à partir de ces derniers. Ces outils supportent les formats standardisés que sont UML, SysML ou MARTE. Le processus d'utilisation de ces outils est défini par les différentes connexions entre les outils de la figure, néanmoins, ces outils sont généraux et non spécifiques à la méthode MOPCOM. En ce sens, ils supportent certes toutes les phases de la méthode MOPCOM, mais ne permettent pas de guider et d'assister l'utilisateur pour suivre correctement le processus défini, ni d'assurer la cohérence sémantique des assemblages des artefacts produits par le processus. Ils ne supportent donc que partiellement le processus et le langage de la méthode MOPCOM. L'aspect multi-utilisateurs de ces outils n'est pas abordé dans [Koudri *et al.* 2008, Aulagnier *et al.* 2009]. L'utilisation d'Eclipse suggère l'utilisation des mécanismes de gestions de révisions, mais les auteurs ne disent rien quant à leurs potentielles utilisations.

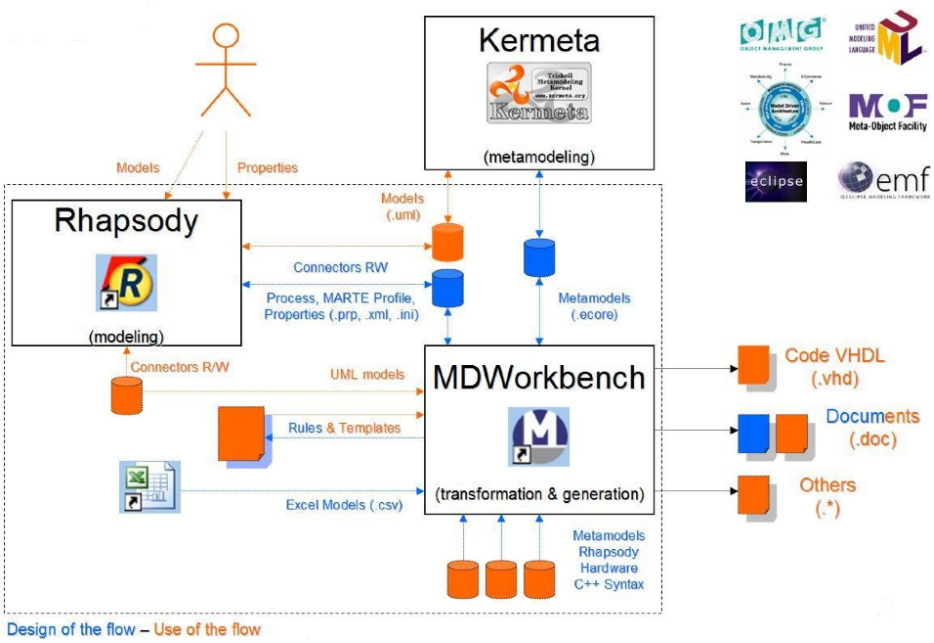


FIGURE 27 – La chaîne d'outils utilisé dans MOPCOM [Aulagnier *et al.* 2009]

Synthèse. Le tableau 12 résume les principales caractéristiques prises en compte par la méthode MOPCOM. Le processus est guidé et rigide, des activités d'analyse sont suggérées afin de valider les différents niveaux de modèles pour passer au niveau suivant. Les autres caractéristiques identifiées ne semblent cependant pas supportées par le processus de la méthode MOPCOM. Le langage MOPCOM étant principalement basé sur SysML et MARTE, nous retenons l'hétérogénéité de MARTE et la composabilité des diagrammes de composants du point de vue structurel. La méthode est supportée par un ensemble d'outil dédié au langage mais non au processus. Cet ensemble ne semble pas dédié à supporter efficacement l'ingénierie et la problématique de gestion de projet n'est pas abordée.

(a) Processus

Guidage	●
Rigidité	◐
Itératif & Incrémental	○
Traçabilité	○
Réutilisabilité	○
Rôles & Responsabilités	○
Parallélisme	◐

(b) Language

Multi-vues	○
Composabilité	◐
Support à l'abstraction	◐
Support à l'hétérogénéité	●
Degré de formalisme	○

Support au processus	◐
Support au langage	○
Support à la gestion de projet	○
Support à la gestion de versions	○

(c) Outils

TABLE 12 – Caractéristiques supportées par la méthode Metropolis

3 Conclusion et positionnement de notre approche

Cette section est une conclusion à l'état de l'art. Elle synthétise les forces et faiblesses des méthodes et approches existantes et nous permet de positionner notre travail de recherche. La première partie résume l'étude précédente des méthodes actuelles, la seconde identifie les faiblesses de ces méthodes et nos propositions pour les combler.

3.1 Synthèse des méthodes actuelles

Le tableau 14 résume les caractéristiques prises en compte par les méthodes de développement de systèmes embarqués actuelles suivant les trois perspectives du processus, du langage et de l'outillage. De manière générale, les méthodes actuelles s'appuient sur la conception de modèles du système à partir desquels du code est généré pour des plates-formes cibles. Toutes les méthodes étudiées permettent de modéliser la partie applicative des systèmes embarqués suivant des approches orientées modèles et parfois composants. La plate-forme d'exécution sur laquelle est implémentée les fonctionnalités décrites dans les modèles d'application est soit injectée statiquement dans les modèles comme c'est le cas de la méthode ACCORD/UML [Gérard 2000], soit prise en compte dans le processus de modélisation du système tel que c'est le cas dans les approches orientées plates-formes, par exemple Metropolis [Sangiovanni-Vincentelli 2007].

Les processus proposés sont tous orientés activités et produits et présentent des caractéristiques générales telles que le caractère itératif, le retour en arrière possible. Ils sont généralement continus, et démarrent à partir de phases plus ou moins abstraites ou informelles telles que la spécification des besoins, et se terminent par des activités d'implémentation qui se traduisent généralement par de la génération de code. Certaines méthodes et approches définissent un langage dédié (par exemple BIP [Basu *et al.* 2011] ou Métropolis [Sangiovanni-Vincentelli 2007]), d'autres s'appuient et étendent des langages déjà éprouvés par la communauté des systèmes embarqués. C'est le cas de MOPCOM [Aulagnier *et al.* 2009] qui s'appuie sur les deux langages SysML et MARTE. Ces langages intègrent généralement différentes perspectives et permettent

Processus	ACCORD / UML	Metropolis	BIP	MOPCOM
Guidage	◐	○	●	●
Rigidité	○	○	●	◑
Itératif & Incrémental	◐	●	◐	○
Traçabilité	○	◐	○	○
Réutilisabilité	○	◐	●	○
Rôles & Responsabilités	○	○	○	○
Parallélisme	○	○	○	◑
Langage				
Multi-vues	●	◐	◐	○
Composabilité	◐	◐	●	◐
Support à l'abstraction	◐	○	◐	◐
Support à l'hétérogénéité	○	◐	○	●
Degré de formalisme	●	◐	●	○
Outils				
Support au processus	◐	◐	●	◐
Support au langage	◐	●	●	○
Support à la gestion de projet	○	○	○	○
Support à la gestion de versions	○	○	○	○

Table 14 – Synthèse des caractéristiques supportées par les méthodes existantes

une certaine forme de composabilité, plus ou moins bien gérée. Du point de vue de l'outillage, les approches sont partagées. Certaines méthodes telles que ACCORD/UML s'appuient sur des outils existants et les personnalisent, d'autres conçoivent des outils dédiés (par exemple BIP), ou encore manipulent et interconnectent plusieurs outils ensemble (par exemple MOPCOM).

3.2 Les faiblesses des méthodes actuelles

Les principales faiblesses des méthodes actuelles résident dans les processus et les outils utilisés. Les processus de développement sont souvent partiellement formalisés et l'aspect de pilotage de projet est souvent inexistant. De plus, ces processus n'exhibent aucun rôle ou responsabilité pour les acteurs. L'outillage quant à lui supporte bien le langage, mais n'est pas connecté à des outils de gestion de projet ou de versions. Cette partie identifie quatre faiblesses majeures dans les méthodes de développement actuelles.

Des processus non complets. Les processus des méthodes étudiées exhibent une démarche continue, généralement descendante dans un enchainement d'activités logique et ordonné et prennent en compte quelques caractéristiques générales. Cependant, ils restent très flous sur ces caractéristiques et n'expliquent pas toujours de quelle manière elles sont prises en compte.

Par exemple, dans ACCORD/UML, le processus est défini comme itératif et incrémental, mais ces deux termes ne sont pas définis, et l'auteur n'explique pas quand démarre une nouvelle itération et de quelle manière elle impacte le reste du processus et le système en cours de développement.

Les définitions des rôles et des responsabilité et des activités sont manquantes dans les processus. L'aspect organisationnel des équipes peut difficilement être traité dans ces cas-là et il est difficile de définir un éventuel parallélisme des activités au sein du processus sans la définition claire des rôles et responsabilités de chacun. Importants dans le cas des développements monolithiques au sein d'une même entreprise, ils deviennent cruciaux au sein des processus distribués entre plusieurs entreprises dans lesquels les relations de sous-traitance doivent être prises en compte. Par exemple, dans [Gérard 2000], Gérard introduit les problématiques liées à la gestion de la sous-traitance, mais le processus proposé n'aborde pas le rôle de sous-traitant et n'y associe donc aucune responsabilité.

La définition d'activité est également manquante dans les processus actuels. Notamment, ils n'expliquent pas quand démarrent ou se terminent les activités, le choix étant le plus souvent laissé au libre arbitre des utilisateurs du processus. Dans certains cas, des analyses et des vérifications opérées par des outils spécialisés laissent suggérer les bornes des activités, mais ce n'est explicité dans aucun des processus étudiés.

Une gestion de projet et une traçabilité non connectées aux modèles de produits. La mesure et le pilotage de l'avancement d'un développement ne peuvent avoir lieu sans un réel couplage entre une gestion de projet, une traçabilité par rapport aux spécifications du systèmes en cours de développement et les modèles produits. D'une part, la traçabilité n'est pas pensée au niveau des processus et d'autre part, la gestion de projet est rarement prise en compte. Cette dernière s'effectue généralement au moyen d'outils de gestion de projets généraux totalement découplés du développement en lui-même. Ainsi, il n'y a pas de réelle connexion entre les développements effectués et leur suivi et un fossé peut se creuser entre les deux rendant inefficaces les efforts de mesure ou de pilotage de l'avancement.

Nous pensons que la gestion de projet devrait, au même titre que la traçabilité, être entièrement exhibée et connectée, par exemple par l'ajout et l'intégration des activités de gestion de projet parmi les activités de développement. Une connexion réelle avec les produits développés permettrait par exemple de concevoir des outils avancés de gestion pour une mesure très précise et fiable des développements et pouvoir coordonner diverses activités d'ingénierie.

Des outils non connectés et non dédiés. Certaines méthodes préconisent l'utilisation d'outils non dédiés et possiblement inter-connectés. C'est le cas par exemple de la méthode MOPCOM [Aulagnier *et al.* 2009] dont le processus est supporté par différents outils (Rhapsody, Kermeta, MDWorkbench, etc.) ou bien ACCORD/UML [Gérard 2000] qui s'appuie sur Objecteering. Cela ne permet de toute évidence pas un suivi des processus de développement et permet des divergences d'utilisation. L'utilisation d'outils divers, fonctionnant possiblement sur des langages différents, introduit une difficulté supplémentaire due à leur interopérabilité. Ces problèmes d'interopérabilité impliquent la définition de nouvelles activités de connexion au sein du processus, alourdissant ce dernier. L'utilisation d'outils non dédiés nécessitent de préalablement personnaliser les outils en question pour supporter efficacement les processus de développement. Dans la plupart des cas, ces personnalisations sont souvent partielles et introduisent une complexité dans la compréhension et l'utilisation des outils au lieu de les faciliter.

La chaîne d'outils BIP présente selon nous un bon usage d'outils dédiés à l'application du processus. Il s'appuie sur un langage unique, le langage BIP pour assurer l'intéropérabilité des outils. Ils n'intègrent cependant par la perspective de gestion de projets.

Il est essentiel que l'outillage soit parfaitement adapté au support du langage, mais aussi du processus afin de ne pas alourdir l'effort de l'utilisateur. L'outil se doit d'être intuitif et dédié. Dans le cas de chaînes d'outils, l'intéropérabilité entre les différents outils doit être être parfaitement gérée et tous les outils de la chaîne doivent être parfaitement connectés.

Pas de processus pour le développement des plates-formes. Si beaucoup de démarches proposées mettent l'accent sur un développement progressif et descendant d'applications, en partant de modèles puis en se rapprochant d'une solution concrète par une succession de raffinement, l'hypothèse est souvent faite que les composants de la plates-formes pré-existent. Ainsi, les plates-formes sont souvent considérées comme de simples assemblages de composants. Par exemple, dans ACCORD/UML, la plate-forme est totalement externe au processus et rien n'indique la façon de l'obtenir. BIP ne définit pas non plus la plate-forme accueillant l'application développée. Les démarches ne sont alors pas consistantes. Rien ne justifie le choix d'un composant plutôt qu'un autre, hormis des considérations purement informelles provenant de l'œil avisé et de l'expérience des concepteurs de plates-formes. Ainsi, il n'est pas possible, à un moment du développement, de sélectionner un composant d'une plate-forme et de tracer son origine, c'est-à-dire d'identifier la raison de sa présence au sein du développement. De plus, sans définir de lien entre le développement de l'application et celui de la plate-forme, il devient alors très difficile de savoir si cette dernière est correctement dimensionnée. Nous pensons que le développement de la plate-forme devrait, au même titre que le développement de l'application suivre une démarche bien définie permettant de justifier son développement.

Synthèse du chapitre

Ce chapitre nous a permis d'identifier et de synthétiser les différentes caractéristiques devant être couvertes par les processus, les langages et les outils des méthodes de développement des systèmes embarqués. Ces caractéristiques sont issues de l'expérience acquise dans l'ingénierie du logiciel et des systèmes d'informations depuis plus de 40 ans et ne sont que trop peu clairement définies pour la modélisation et la formalisation de méthodes pour les systèmes embarqués. L'étude de quelques méthodes et approches de développement de systèmes embarqués a permis de s'interroger sur la couverture de ses caractéristiques. Il en résulte une forte couverture des langages, mais une faible couverture des processus. Quant aux outils, ils ne sont généralement vus que comme support et ce sont bien souvent des outils génériques, proposant des personnalisations incomplètes.

La dernière section nous a permis d'identifier les forces et faiblesses des méthodes et approches étudiées. En particulier, nous avons identifié quatre faiblesses (processus non complets, gestion de projet et traçabilité non connectées aux modèles de produits, outils non connectés et non dédiés, faiblesse des processus pour le développement des plates-formes) importantes qui nous permettrons de positionner nos contributions. Cette dernière section clôt cette seconde partie du mémoire et sert d'amorce à l'introduction de notre proposition.

Deuxième partie

Proposition

Liste des chapitres

Introduction à la contribution

La seconde partie de ce mémoire présente **notre contribution**. Elle est structurée en six chapitres :

Chapitre 5 (Présentation générale de la méthode $\langle$HOE$\rangle^2$). Ce chapitre introduit brièvement la méthode $\langle$HOE$\rangle^2$ et en délimite les différentes spécificités pour aborder la problématique identifiée.

Chapitre 6 (Description détaillée de la méthode $\langle$HOE$\rangle^2$). Ce chapitre détaille la formalisation du processus de la méthode et du langage de modélisation. Il présente une version restreinte du processus ne prenant pas en compte la composition des plates-formes.

Chapitre 7 (Systèmes embarqués et composition de plates-formes). Ce chapitre présente une extension du processus et du langage afin de prendre en compte la composition de plates-formes au sein de la méthode $\langle$HOE$\rangle^2$.

Chapitre 8 (Gestion de projet et traçabilité couplées dans le processus). Ce chapitre aborde l'intégration de la traçabilité et de la gestion de projet. La contribution présentée permet d'une part d'assurer la traçabilité entre les modèles conçus et les activités de modélisation, et d'autre part d'offrir au chef de projet des outils efficaces pour mesurer et piloter l'avancement du développement et organiser ses équipes de développement.

Chapitre 9 (CanHOE2, un atelier de développement dédié). Ce chapitre présente les développements réalisés pour l'outillage de la méthode $\langle$HOE$\rangle^2$. L'outil réalisé se nomme CanHOE2 (pour *CANonical* $\langle$*HOE*$\rangle^2$). Il est dédié et permet le support du langage ainsi que du processus. Il offre un environnement de développement multi-utilisateurs et intègre des vues de gestion de projet à l'attention du chef de projet.

Chapitre 10 (Validation). Ce chapitre présente une étude de cas permettant de vérifier certaines propriétés de la méthode. Dans ce chapitre, nous présentons d'une part comment la méthode $\langle$HOE$\rangle^2$ aborde la transformation des modèles de l'application indépendants de la plate-forme en des modèles spécifiques à cette dernière, et d'autre part comment, à partir de ces modèles spécifiques, nous pouvons générer du code pour ces plates-formes cibles à l'aide de générateurs de code spécifiques.

Chapitre 5

Présentation générale de la méthode ⟨HOE⟩²

Liste des sections

Afin de répondre aux problématiques identifiées dans les chapitres précédents qui sont non traitées ou faiblement traitées par les méthodes actuelles, nous introduisons la méthode ⟨HOE⟩², pour *Highly Heterogeneous Object-Oriented Efficient Engineering*, qui est dédiée au développement de systèmes embarqués.

Ce chapitre est structuré ainsi : la section 1 présente d'une manière générale la méthode ⟨HOE⟩² ; la section 2 introduit le processus de développement de la méthode ⟨HOE⟩² ; la section 3 exhibe l'aspect collaboratif du processus et ses bénéfices en termes de gestion de projet et de simplification de développement ; enfin, la section 4 expose le caractère fractal du processus et présente ses avantages pour la composition de plates-formes complexes. La gestion de projet intégrée et l'outillage sont respectivement présentés dans les sections 5 et 6.

1 Présentation générale

La méthode ⟨HOE⟩² – pour *Highly Heterogeneous Object-Oriented Efficient Engineering*, ou *ingénierie efficace orientée objet hautement hétérogène* cible le développement de systèmes embarqués. Elle partage des racines communes avec des cycles de développement classiques est les processus unifiés USDP et RUP [Ivar Jacobson *et al.* 1999, IBM 2003]. Elle est inspirée des cours réalisés par l'équipe des systèmes d'information à l'Institut universitaire de technologie Grenoble 2 [OMGL 1997] et du Processus de développement de systèmes industriels (PDSI) [Rygaert 2002]. ⟨HOE⟩² vise à la fois le développement logiciel (applications, middleware, systèmes d'exploitation, etc.) et le développement matériel (ASIC, FPGA, etc.).

Tout le long de ce chapitre, la méthode sera présentée globalement selon trois composantes : processus, langage et outil. ⟨HOE⟩² désignant à la fois la méthode, le processus et le langage, nous préciserons systématiquement ce qui est désigné par le terme ⟨HOE⟩².

Le processus de développement. Le processus ⟨HOE⟩² suit une approche descendante et est accompagné de la formalisation de ses acteurs et des interactions entre ces derniers. Il permet le développement conjoint et parallèle d'une application d'un système embarqué et

de sa plate-forme. Il répond au besoin exprimé dans les chapitres précédents de posséder un processus formalisé, rigide, guidant l'équipe de développement tout le long du développement des systèmes embarqués et facilitant la parallélisation des tâches au sein de l'équipe projet.

Le processus ⟨HOE⟩² sera progressivement présenté dans ce chapitre afin de mettre en évidence : 1) les quatre phases de développement, définissant le processus canonique ; 2) les aspects collaboratifs menant à un développement conjoint et parallèle d'une application et de sa plate-forme et 3) son caractère fractal, permettant de concevoir des plates-formes complexes par composition de plates-formes plus simples.

Le langage ⟨HOE⟩². Le langage ⟨HOE⟩² est structuré en deux grandes parties : la partie *produit* et la partie *gestion de projet*. Le langage *produit* repose sur un concept de base qui est celui d'*objets* connectés et communicant par échange de messages. Dans ⟨HOE⟩², un objet permet de représenter uniformément un composant applicatif ou de la plate-forme et ce, quelque soit le niveau d'abstraction ou la technologie (logicielle ou matérielle) considérée. À ce titre, en plus des aspects multi-vues et composition habituellement traités, le langage ⟨HOE⟩² offre le support à l'abstraction et à l'hétérogénéité nécessaires pour la conception de systèmes embarqués.

La partie *gestion de projet* du langage peut être vue comme un greffon qui étend le langage *produit* ⟨HOE⟩². Il répond essentiellement à la problématique identifiée dans le chapitre 4 sur le manque de connexion des éléments de gestion de projet aux modèles de produits développés.

Le langage est graphique et *dédié*. Il a été proposé sous la forme d'un langage spécifique de domaine DSL [1] étendant et restreignant UML. Un DSL permet de définir un langage adapté à un domaine. Le choix de concevoir un DSL comme extension d'UML s'argumente de part le nombre de concepts similaires présents dans la langage UML et par la simplicité d'implémentation dans des outils UML. La syntaxe concrète est également une adaptation de celle du langage UML.

L'outillage CanHOE2. CanHOE2 est développé sous la forme de greffons Eclipse et est dédié au support de la méthode ⟨HOE⟩². Il supporte à la fois le processus et le langage tout en proposant des fonctionnalités pour favoriser une gestion de versions efficace et une intégration forte de la gestion de projet connectée aux modèles produits. La figure 28 illustre l'outillage CanHOE2. Il possède un explorateur pour naviguer au sein des modèles développés, une zone d'édition graphique pour modéliser le système selon une vue particulière et des vues de gestion de projet telles que la vue des tâches pour favoriser le développement collaboratif. L'outillage sera détaillé dans le chapitre 9.

Conclusion préliminaire et rappel des faiblesses adressées. Dans cette section, nous avons introduit les trois composantes de la méthode ⟨HOE⟩². Les prochaines sections viseront à présenter les points originaux de la méthode afin d'adresser les faiblesses identifiées dans le chapitre 4 : processus non complets et n'adressant pas le développement de la plate-forme, gestion de projet inexistante ou déconnectée des produits développés, manque d'outils dédiés. En particulier, nous montrerons dans les prochaines sections que, en mettant essentiellement l'accent sur la présentation du processus, ⟨HOE⟩² vérifie les propriétés suivantes : processus complet, rigide, offrant un support au parallélisme, à la traçabilité et au développement collaboratif, prise en compte du développement de la plate-forme, outil dédié supportant le processus et le langage, incluant la gestion de projet et offrant un support efficace à la gestion de versions.

1. En anglais, Domain Specific Language (DSL).

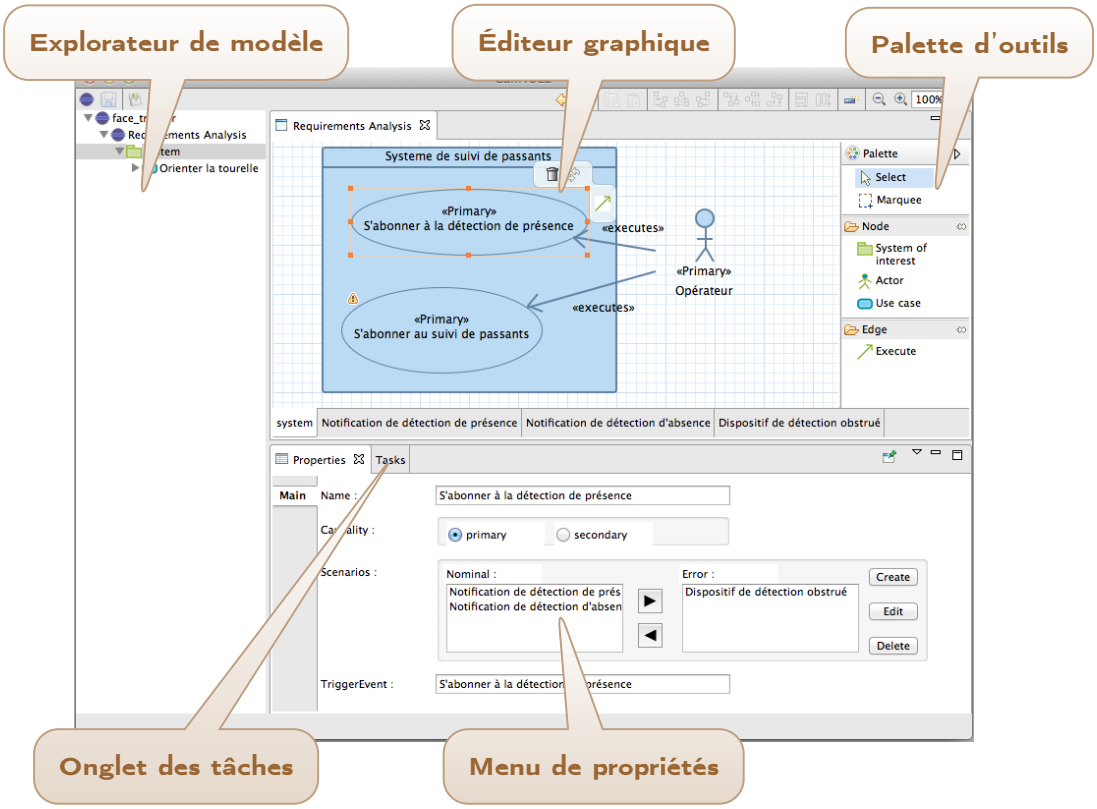


FIGURE 28 – CanHOE2, outillage canonique de ⟨HOE⟩²

2 Un processus canonique pour le développement de systèmes

La méthode ⟨HOE⟩² se distingue des méthodes et approches étudiées dans le chapitre 4 de par son processus complet, avec l'identification claire des phases et des activités durant ces phases, la définition des acteurs du processus [2], les produits développés.

Le processus canonique ⟨HOE⟩² permet le développement d'un *système considéré*. L'INCOSE [3] définit un SOI [4] comme un « *système dont le cycle de vie est sous considération* » [INCOSE 2011]. Dans notre cas, il peut correspondre à l'application ou la plate-forme d'un système embarqué. Ainsi, le processus permet de concevoir l'une et/ou l'autre des deux composantes formant un système embarqué.

La figure 29 présente le processus canonique ⟨HOE⟩². Il suit une approche descendante, orientée activités et produits. Les produits développés le long du processus sont des modèles représentant les aspects fonctionnels, dynamiques et structurels d'une application ou d'une plate-forme. Le processus est divisé en quatre phases classiques de développement que nous

2. La définition des acteurs sera présentée dans la section 3.
3. International Council of Systems Engineering [INCOSE 2014]
4. En anglais, System of Interest (SOI).

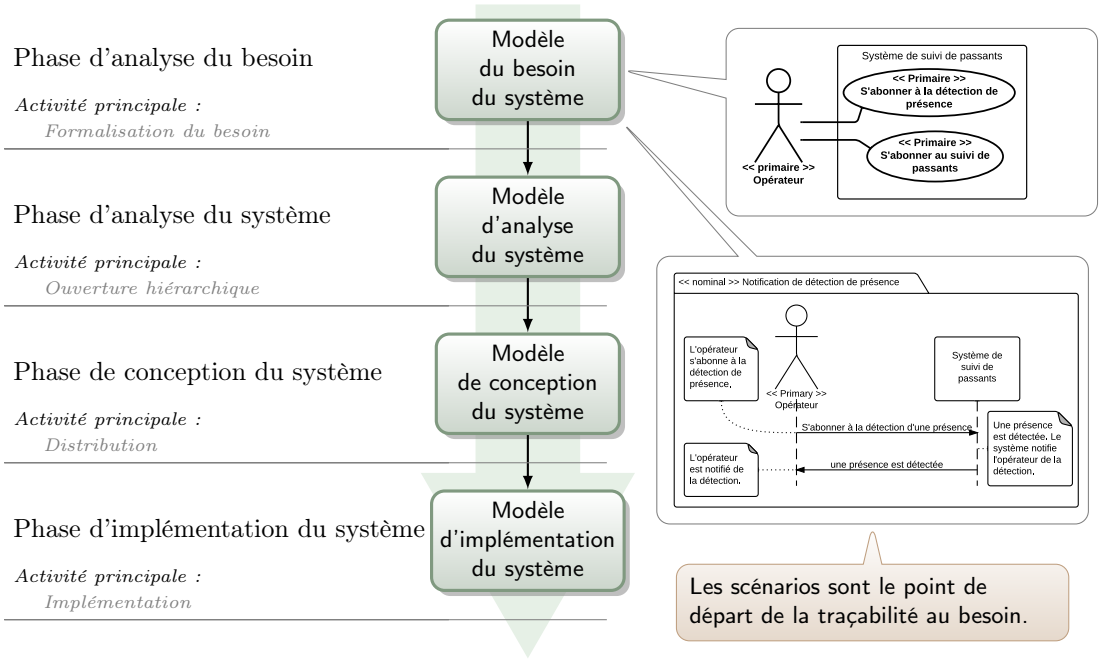


FIGURE 29 – Le processus canonique de la méthode ⟨HOE⟩²

retrouvons au sein de RUP [IBM 2003] ou d'USDP [Ivar Jacobson *et al.* 1999] : *phase d'analyse du besoin du système*, *phase d'analyse du système*, *phase de conception du système* et *phase d'implémentation du système*. Lors de chacune des phases, une activité de modélisation est réalisée : *formalisation du besoin*, *ouverture hiérarchique*, *distribution* et *implémentation*. Nous pouvons retrouver les deux premières activités dans les cycles de développement logiciel traditionnels, les deux dernières sont spécifiques au développement des systèmes embarqués car elles font le lien entre le développement applicatif et celui de la plate-forme. Ce lien sera détaillé dans le chapitre 6.

À chacune des phases, le modèle produit est un enrichissement des modèles produits issus des phases amont (cf. fig. 30). Il est automatiquement initialisé à partir du modèle précédant. Une fois initialisé, des tâches de modélisation permettent de raffiner le modèle. Par exemple, le modèle d'analyse est obtenu par enrichissement du modèle d'analyse du besoin par l'activité d'ouverture hiérarchique lors de la phase d'analyse.

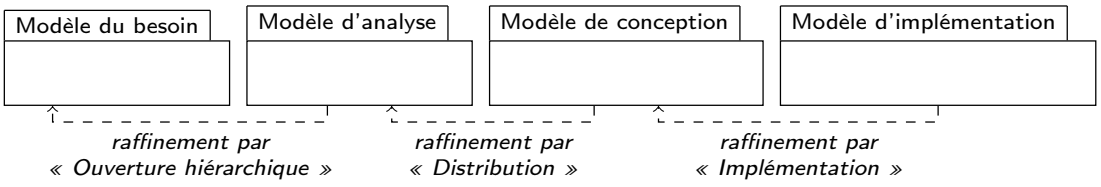


FIGURE 30 – Enrichissement des modèles ⟨HOE⟩²

Le processus ⟨HOE⟩² est itératif et guidé par la largeur du besoin couvert ou restant à couvrir. Deux types d'itération sont possibles : *locales* au niveau de chacune des phases ou *globales* au niveau du processus entier. Une itération locale permet de préciser un besoin identifié dans la première phase du processus alors qu'une itération globale permet d'étendre la couverture des besoins. Le besoin se modélise au moyen de cas d'utilisation et de scénarios. Toutes les tâches de modélisation découlent d'un choix préalable d'un besoin à satisfaire. De ce fait, le processus ⟨HOE⟩² assure la traçabilité des modèles produits par rapport au besoin initialement formulé. Cette partie sera plus longuement expliquée dans la section 5.

En résumé

✓ *Un processus canonique, rigide et parfaitement guidé*
✓ *Une traçabilité connectée aux modèles de produits*

3 Un processus collaboratif pour le développement de systèmes embarqués

Dans la section précédente, nous avons présenté les premières caractéristiques du processus canonique de la méthode ⟨HOE⟩² pour le développement d'une application ou d'une plate-forme.

Dans les cycles de développement classiques, notamment le cycle en Y [Luiz Fernando Capretz 2005, Kienhuis *et al.* 2002], le développement de la branche fonctionnelle est obtenu par itérations successives (locales ou globales) tandis que la conception de la plate-forme n'est pas prise en compte et relève le plus souvent d'un processus externe. Les composants techniques sont alors « *pris sur étagère* ». Cette approche a plusieurs points faibles. D'une part, la plate-forme n'étant pas modélisée au même titre que l'application, il n'y a aucune traçabilité permettant d'assurer qu'un composant de la plate-forme répond bien à un besoin exprimé. D'autres part, ces composants sont souvent peu détaillés, très rarement en termes de comportements et n'explicitent que trop rarement la manière dont une application peut être déployée.

Le principal apport du processus ⟨HOE⟩² est de considérer le développement de la branche technique (la plate-forme) [5] au même titre que la branche fonctionnelle (l'application). Les objets de la plate-forme sont alors obtenus par le même procédé que les objets de la plate-forme. La figure 31 étend la présentation initiale (cf. fig. 29) du processus canonique ⟨HOE⟩² en l'employant simultanément pour le développement de l'application hébergée sur sa plate-forme, les deux flots formant le système embarqué en cours de conception.

La séparation effectuée entre les deux flots apporte de nombreux avantages. Le développement peut être distribué sur les deux branches très tôt dans le processus, favorisant le développement en parallèle. Pour cela, nous faisons le choix, lors de l'activité de formalisation du besoin, de distribuer le besoin sur les deux branches, selon leur appartenance. Cela contribue à séparer deux métiers différents. Ainsi, les spécifications relevant de l'application contribuent à alimenter le modèle de besoin de l'application tandis que les spécifications relevant de la plate-forme contribuent à alimenter celui de la plate-forme. De cette façon, nous obtenons une

5. Dans la figure 31, nous ne présentons volontairement pas les deux derniers niveaux de modèle de la plate-forme. Ils permettront d'introduire la section suivante.

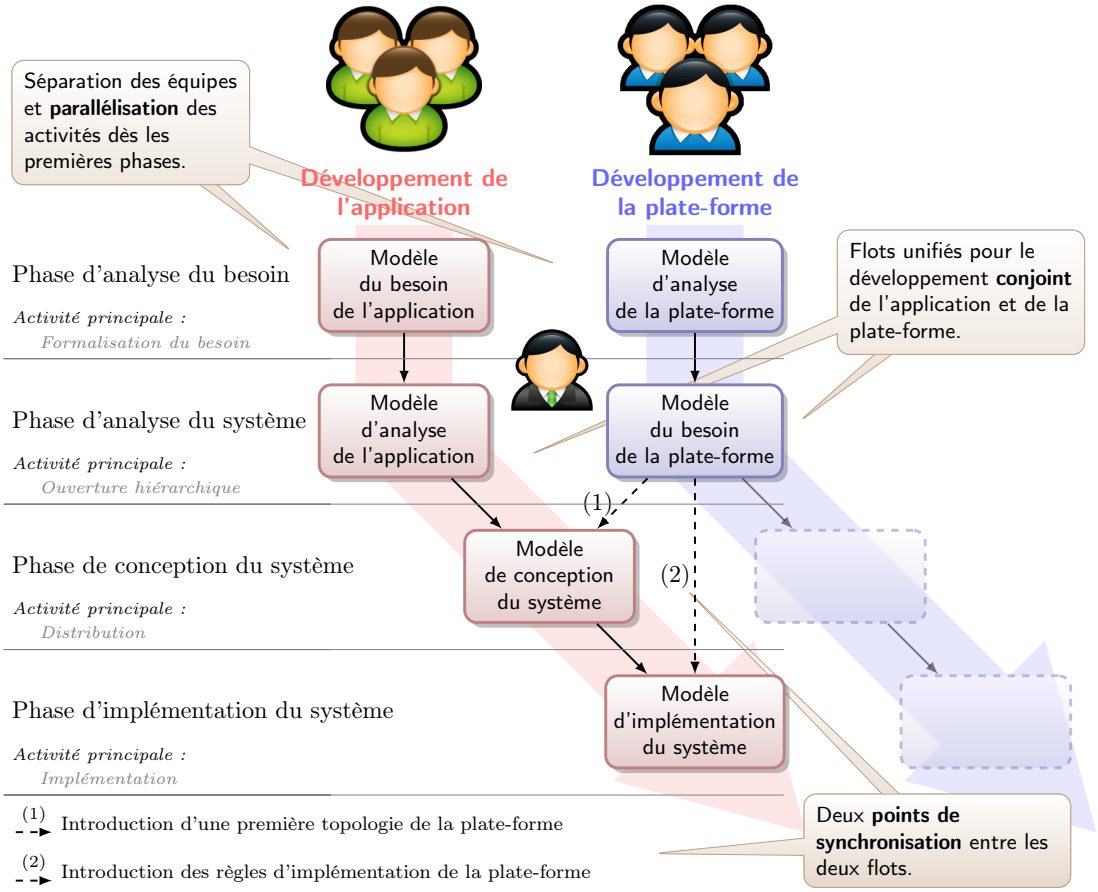


FIGURE 31 – Processus collaboratif pour le développement de systèmes

indépendance dans le développement des deux flots depuis la première phase du processus. Cela a pour bénéfice d'augmenter l'efficacité de développement en exhibant fortement le parallélisme qui constitue l'un des points faibles des méthodes étudiées dans les chapitres précédents.

Contrairement au modèle en Y, dans lequel les deux branches convergent en une seule à partir d'un unique point de synchronisation, le processus ⟨HOE⟩² se démarque en offrant le maximum d'indépendance entre les deux branches de développement, avec deux points de synchronisation clairement identifiés. La figure 31 exhibe les interactions entre les deux flots en deux points de convergence, permettant d'introduire graduellement la plate-forme dans le flot de développement de l'application.

La figure 31 illustre également les acteurs du processus et les points d'interaction entre ces derniers. Trois acteurs sont mis en évidence : les développeurs applicatifs, les développeurs de la plate-forme et le chef de projet. Les deux points de synchronisation constituent un point fort dans la méthode ⟨HOE⟩². Alors que les autres méthodes se contentent de modéliser les allocations des ressources de la plate-forme pour chaque objet de l'application, aucune d'elle ne fournit explicitement des règles d'implémentation des objets sur les ressources. Or, il s'agit d'un point crucial et il n'existe généralement pas une solution unique qui détermine comment

sont implémentés les objets sur la plate-forme. Aussi, nous avons fait le choix d'introduire la plate-forme en deux temps : 1) dans un premier temps sa topologie, c'est-à-dire ses ressources sur lesquelles les objets de l'application vont pouvoir être alloués et ses périphériques accessibles et 2) dans un second temps les règles d'implémentation permettant de définir précisément la manière dont les objets seront implémentés.

Enfin, une dernière originalité réside dans la seconde branche du processus. Le modèle d'analyse de la plate-forme suffit à lui seul à définir entièrement un modèle d'implémentation de l'application sur la plate-forme. Le développement de la plate-forme n'est pas terminé pour autant et suit le même flot (passage en conception puis en implémentation de la plate-forme) que l'application. Cet aspect permet d'introduire le caractère fractal du processus qui sera présenté dans la section suivante.

En résumé

- ✓ *Support à la collaboration et aux développements parallèles*
- ✓ *Développements uniformes de la plate-forme et de l'application*
- ✓ *Points de synchronisation entre les développements clairement définis*
- ✓ *Un langage commun pour les deux flots, facilitant la synchronisation et le travail collaboratif*

4 Un processus fractal pour le développement de plates-formes complexes

Nous avons présenté dans la section précédente un processus de développement collaboratif sur deux branches distinctes mais interagissant sur deux points de synchronisation clairement définis. Contrairement aux systèmes d'information où le développement de la plate-forme est rarement considéré, cette dernière étant souvent considérée comme idéale et possédant des ressources illimitées, l'ingénierie des systèmes embarqués nécessite qu'une plus grande attention soit portée au développement de la plate-forme, cette dernière étant limitée en termes de ressources, d'autonomie, et faisant le pont avec l'environnement physique. Le processus permet le développement de la plate-forme dans un flot de conception similaire à celui de l'application.

Dans la section précédente, nous évoquions le fait que la plate-forme était développée selon le même flot de conception que l'application. Afin de pouvoir expliquer comment sont obtenus les modèles de conception et d'implémentation de la plate-forme, nous étendons le processus sur deux branches précédemment illustré par la figure 31 en un processus *fractal* et *récursif*.

Du point de vue du chef de projet, la notion d'application ou de plate-forme devient relative. Dans la figure 32, P_1 est une plate-forme dès l'instant où le système considéré est l'application hébergée par P_1. Maintenant, si l'on prend P_1 comme le système considéré, il est perçu comme la surcouche applicative de la plate-forme P_2 destinée à héberger l'application. Par analogie, le système d'exploitation est une couche applicative sur une plate-forme physique offrant des fonctionnalités d'ordonnancement pour différentes applications. Pour ces dernières, le système d'exploitation est alors la plate-forme sur laquelle elles s'exécutent.

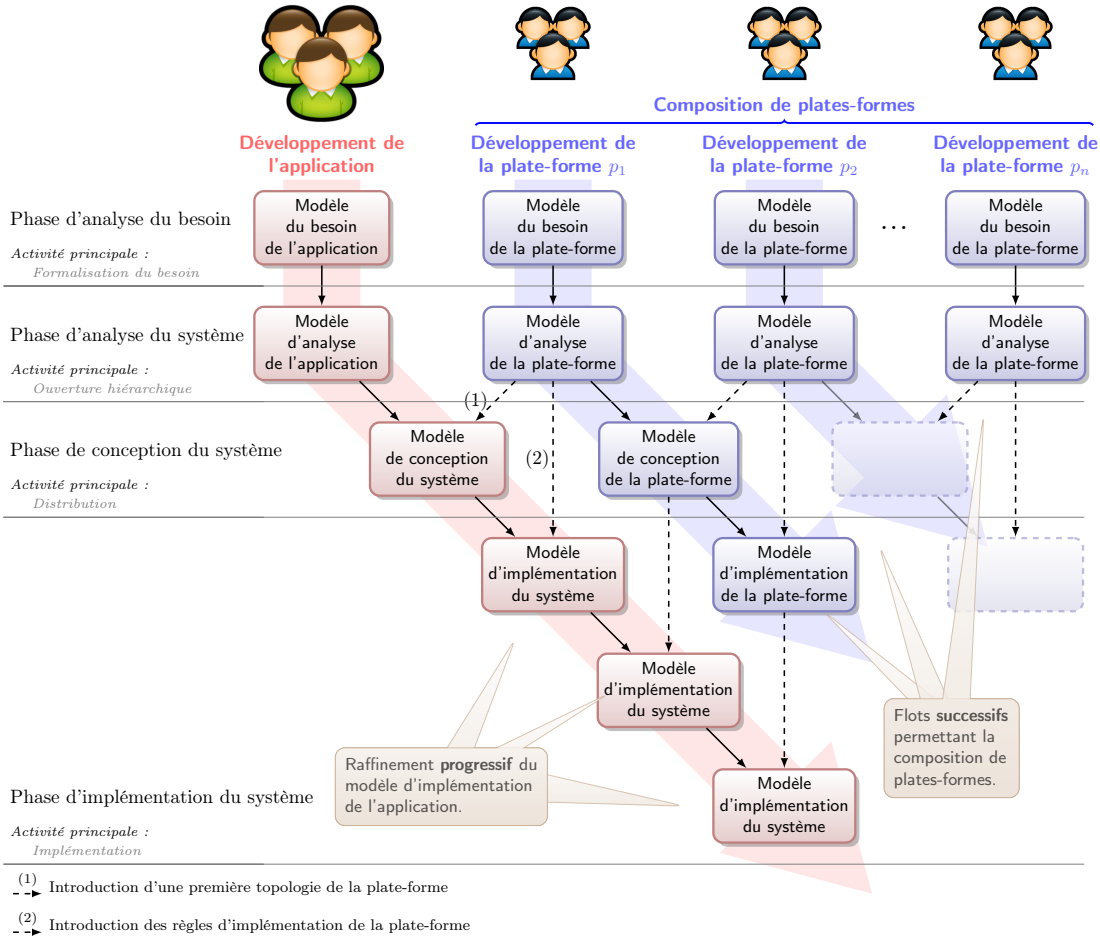


FIGURE 32 – Processus fractal pour la composition de plates-formes

La figure 32 illustre ce processus fractal. Elle fait apparaitre une branche pour le développement de l'application et plusieurs branches pour le développement de la plate-forme. Selon ce processus, le développement de la plate-forme est obtenu par *composition*. Ainsi, le processus ⟨HOE⟩² permet non seulement le développement conjoint de la plate-forme et de l'application, mais permet également de simplifier le développement de plates-formes complexes en les scindant en plusieurs développements largement découplés. Cela accentue le parallélisme possible entre les équipes de développement.

Le processus présenté se termine lorsque la plate-forme la plus à droite est considérée comme *terminale*. Le développement d'une plate-forme terminale diffère des autres. Le flot n'est constitué que des deux premières phases du processus (phases du besoin et d'analyse de la plate-forme). Elle est fournie avec les règles de génération de code. Ces règles permettent de générer, à partir d'un modèle d'application implémenté sur une plate-forme finale, le code pour cette plate-forme.

Le processus fractal amène une nouvelle dimension dans le développement du système. Dans la section précédente, nous évoquions que seules les données collectées dans le modèle d'analyse de la plate-forme étaient suffisantes pour construire un modèle d'implémentation de l'application. Puisque la plate-forme poursuit son développement sur les phases aval du processus, il en va de même pour l'application dont le modèle d'implémentation est progressivement et *automatiquement* raffiné. Le raffinement automatique est rendu possible dans la mesure où une intervention humaine a permis de construire le premier modèle d'implémentation de l'application sur le modèle d'analyse de la plate-forme, ce dernier ayant lui-même été raffiné dans les phases de conception et d'implémentation lors d'activités d'ingénierie.

Afin de mieux illustrer le raffinement progressif de l'application sur les différentes plates-formes, la figure 32 (tout comme la figure 31 précédemment) présente le modèle d'implémentation de l'application à la verticale de la plate-forme sur laquelle elle est implémentée. Ainsi, le dernier modèle visible sur la figure repose sur la plate-forme p_2. Dans la mesure où cette dernière sera à terme elle-même implémentée sur une autre plate-forme p_n, le modèle d'implémentation final de l'application sur la plate-forme se trouve à la verticale de la plate-forme tout à droite.

En résumé, la composition étend considérablement le processus de développement et lui confère de nombreux avantages. Il permet une distribution plus fine et plus structurée des équipes et des développements d'une plate-forme. Il permet également de monter progressivement en niveau d'abstraction pour simplifier le raisonnement et le développement. Enfin, il est possible d'introduire graduellement les contraintes physiques sur différents niveaux d'abstraction. En outre, le langage commun pour le développement de l'application et de la plate-forme facilite les points de synchronisation entre chaque branche. La composition sera détaillée dans le chapitre 7.

En résumé

- ✓ *Prise en compte du développement de la plate-forme dans le processus*
- ✓ *Possibilité de décomposer la plate-forme pour renforcer le parallélisme*
- ✓ *Distribution des équipes de développement sur chaque branche*
- ✓ *Support à l'abstraction pour les plates-formes*

Nous avons jusqu'ici présenté la manière dont la méthode $\langle \mathsf{HOE} \rangle^2$ aborde de nombreuses problématiques en nous basant sur des descriptions succinctes du processus et du langage. Les problématiques actuellement abordées sont le besoin de posséder un processus rigide et parfaitement défini, prenant en compte le développement de plates-formes, plus ou moins complexes tout en maitrisant leur complexité au moyen de la composition de plates-formes. Le processus est collaboratif et ses acteurs sont clairement identifiés. Il offre un support au parallélisme ainsi qu'à l'abstraction. Les flots proposés sont indépendants, avec des points de synchronisation clairement définis, la synchronisation étant facilitée par l'utilisation d'un langage commun. Enfin, le langage et le processus permettent de connecter la traçabilité des modèles de produits développés. Les deux sections suivantes présentent plus en détail la gestion de projet et l'outil dédié, afin de répondre aux dernières problématiques.

5 Gestion de projet intégrée et connectée aux modèles de produits

La gestion de projet est très rarement traitée au sein des méthodes de développement de systèmes embarqués et constituent une faiblesse de ces méthodes. Elle est généralement assurée par des outils externes et découplés des modèles de produits. Sans connexion directe, de tels outils ne permettent pas une mesure précise de l'état d'avancement d'un projet, et n'offrent aucune possibilité de pilotage efficace.

Dans ⟨HOE⟩², la gestion de projet est adressée par le processus, ainsi que le langage et est supportée par l'outil dédié à la méthode, comme nous le verrons dans la section suivante. La figure 33 illustre le dialogue entre le chef de projet et un développeur lors de l'affectation d'une tâche. Le chef de projet planifie la tâche à réaliser et l'assigne à un développeur. Il envoie la tâche ainsi qu'un modèle partiel au développeur. Le modèle envoyé est une version partielle du modèle global, mais néanmoins auto-suffisante pour l'accomplissement de la tâche au développeur. Le développeur peut démarrer l'activité de modélisation dès sa réception.

Une fois la tâche effectuée par le développeur, ce dernier renvoie au chef de projet le modèle partiel qu'il a modifié, afin que le chef de projet puisse le valider ou le rejeter. La figure 34 illustre ce second échange. Afin de pouvoir valider et l'intégrer dans le modèle global, ou rejeter le modèle, la tâche du chef de projet consiste préalablement à *rejouer les scénarios* pour valider que les modifications apportées par le développeur répondent bien aux besoins à satisfaire et ne remet pas en cause les besoins existants. Le *rejoué de scénario* consiste à réinterpréter un ou plusieurs scénarios définis au niveau du besoin afin de voir en quoi le rajout ou la suppression d'éléments dans les modèles influence la satisfaction du besoin initial. Cette activité peut être automatisée et contribue grandement à améliorer l'efficacité d'ingénierie.

Ces deux activités sont indépendantes. Cela permet au chef de projet de pouvoir affecter de nouvelles tâches à d'autres développeurs sans devoir attendre que la première tâche affectée ait été réalisée. Cela favorise également le travail collaboratif et offre un support au parallélisme. À

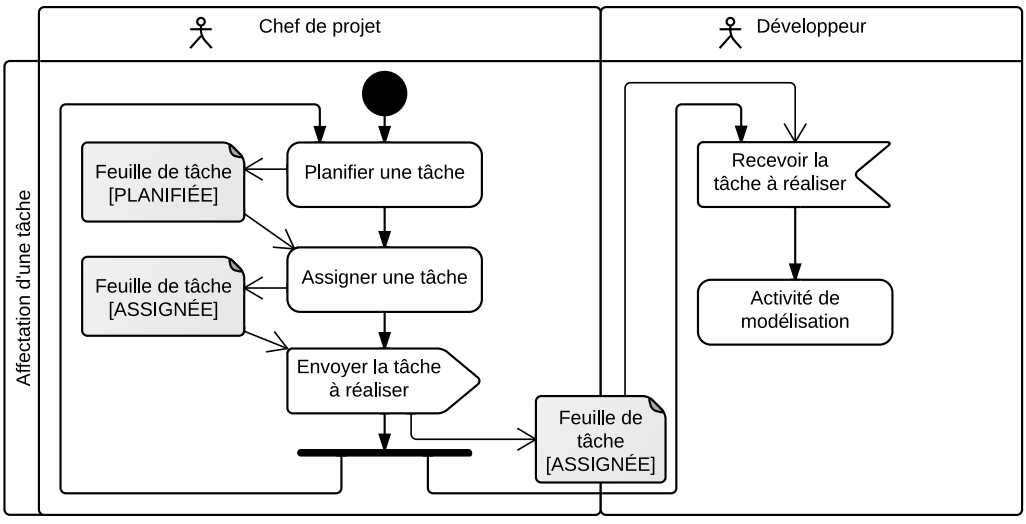


FIGURE 33 – Assignation d'une tâche et activité de modélisation

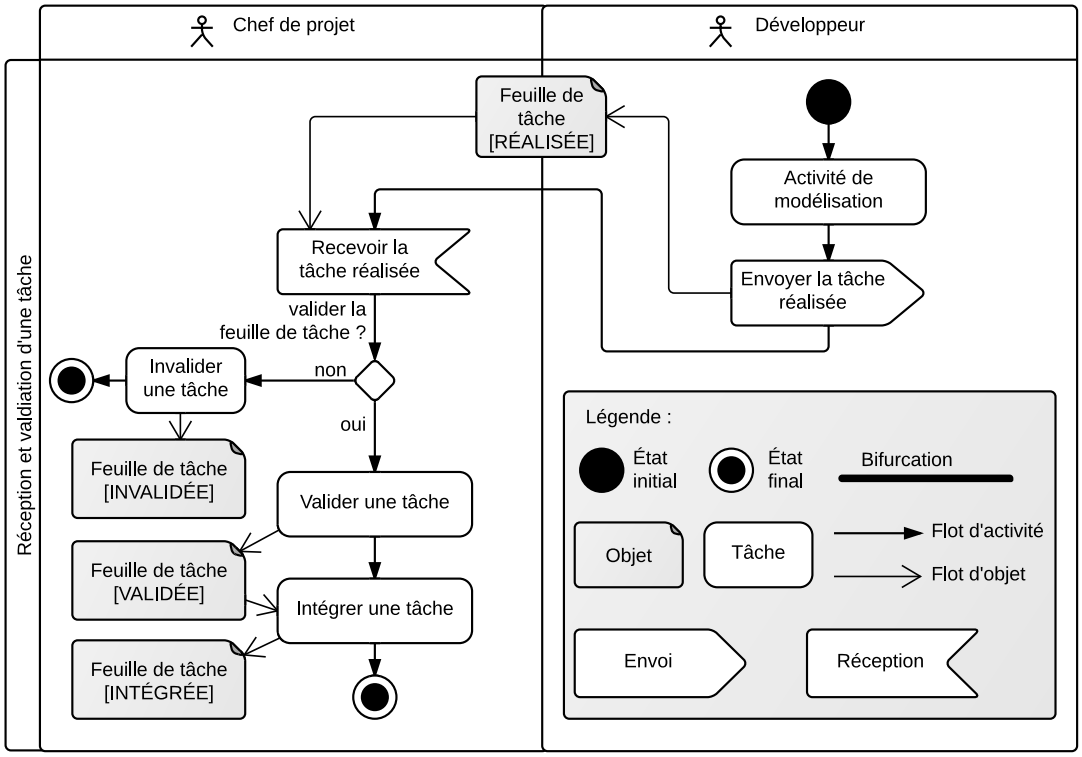


FIGURE 34 – Réalisation d'une tâche et intégration du résultat de modélisation

tout moment, le chef de projet peut voir quelles sont les tâches affectées. Ces données permettent d'alimenter des vues de gestion de projet telles qu'un diagramme de Gantt par exemple.

Afin de pouvoir supporter la gestion de projet au sein de la méthode, une partie du langage $\langle \mathsf{HOE} \rangle^2$ est dédiée à la définition des briques élémentaires pour l'exprimer. Entre autre, les concepts de *tâche* et de *feuille de tâche* sont définis. La figure 35 est un modèle général d'une *feuille de tâche* (instance particulière d'une tâche affectée à un développeur). Dans $\langle \mathsf{HOE} \rangle^2$, une feuille de tâche est caractérisée par différentes propriétés : le développeur auquel elle est affectée, le modèle partiel entrant, les scénarios à satisfaire et un ensemble de données (dates de début et fin, description, nom, etc.). En cours de réalisation, elle est définie par le nombre de scénarios réalisés et le modèle en cours de développement. Une fois la tâche achevée, c'est-à-dire lorsque tous les scénarios à satisfaire sont réalisés, elle est retournée au chef de projet qui peut la valider et l'intégrer, ou la rejeter. Ainsi, le chef de projet a une vue de l'ensemble des tâches et peut tracer chaque élément du modèle de produits pour découvrir la tâche qui l'a modifié. Cela contribue à connecter la gestion de projet aux modèles de produits.

Afin d'apporter une granularité plus fine, l'ensemble de tâches a été identifié et spécialisé pour chacune des phases. Par exemple, lors de la phase d'analyse du système, deux tâches sont définies : la tâche d'*ouverture hiérarchique* d'un objet du système et la tâche de *complément d'ouverture*. Ainsi, pour chacune des tâches, le modèle partiel est spécialisé afin de définir précisément les éléments du modèle affectés par la tâche. Nous détaillerons l'ensemble de ces tâches dans le chapitre 6.

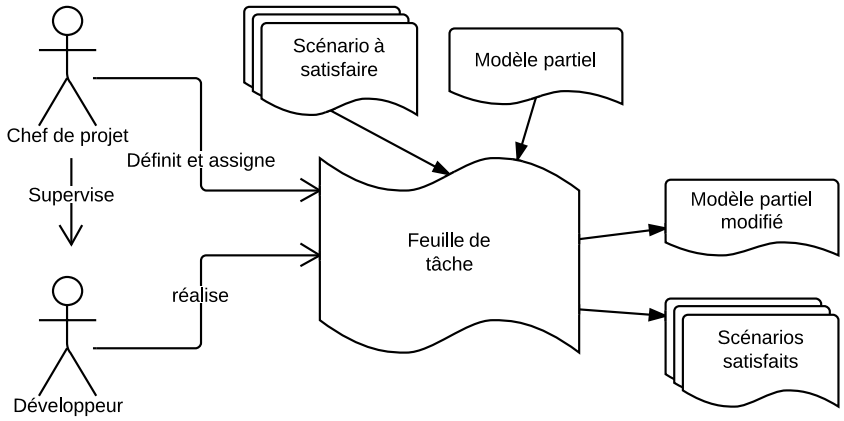


FIGURE 35 – Définition d'une feuille de tâche

EN RÉSUMÉ

- ✓ *Spécialisation des tâches de gestion de projet à chaque phase, accentuant la traçabilité*
- ✓ *Un modèle de tâches permettant de lier la gestion de projet aux modèles produits*
- ✓ *Une ingénierie efficace rendue possible par des activités d'intégration et de validation*
- ✓ *Guidage dans les activités de collaboration*

6 Outil dédié au support de la méthode ⟨HOE⟩²

L'outillage dédié ⟨HOE⟩², nommé CanHOE2 (pour Canonical ⟨HOE⟩²) a été introduit dans la section 1 et illustré par la figure 28. CanHOE2 a été développé sur la plate-forme Eclipse sur la base des outils de modélisation et de transformation de modèles de la plate-forme (éditeurs graphiques de modèles, outils de transformation de modèles, de génération de code).

L'outil CanHOE2 vise à répondre aux problématiques identifiées dans les chapitres précédents en offrant non seulement un support pour guider les équipes de développement autour de la méthode ⟨HOE⟩², mais également un support à la gestion de projet intégrée ainsi qu'un support à la gestion de versions.

La gestion de rôles et des différents acteurs est gérée au sein de l'outil par l'utilisation de différentes vues, ou *perspectives*. Deux perspectives sont proposées. L'une d'elle est dédiée au développement selon ⟨HOE⟩² (cf. fig. 28) et accessible aux développeurs. Elle permet à chaque développeur de réaliser une activité qui lui a été préalablement affectée par le chef de projet. Le développeur possède une vue restreinte du projet, focalisée sur l'accomplissement de l'activité qui lui a été confiée. À la différence du chef de projet, Il ne possède pas une vision globale du développement courant du projet, seulement une vision restreinte qui correspond à sa tâche.

Une seconde perspective, dédiée au chef de projet, lui permet de superviser les développements. La perspective du chef de projet (cf. fig. 36) possède une vue d'ensemble du projet et de toutes les activités en cours et passées. La zone d'édition graphique se résume à une zone

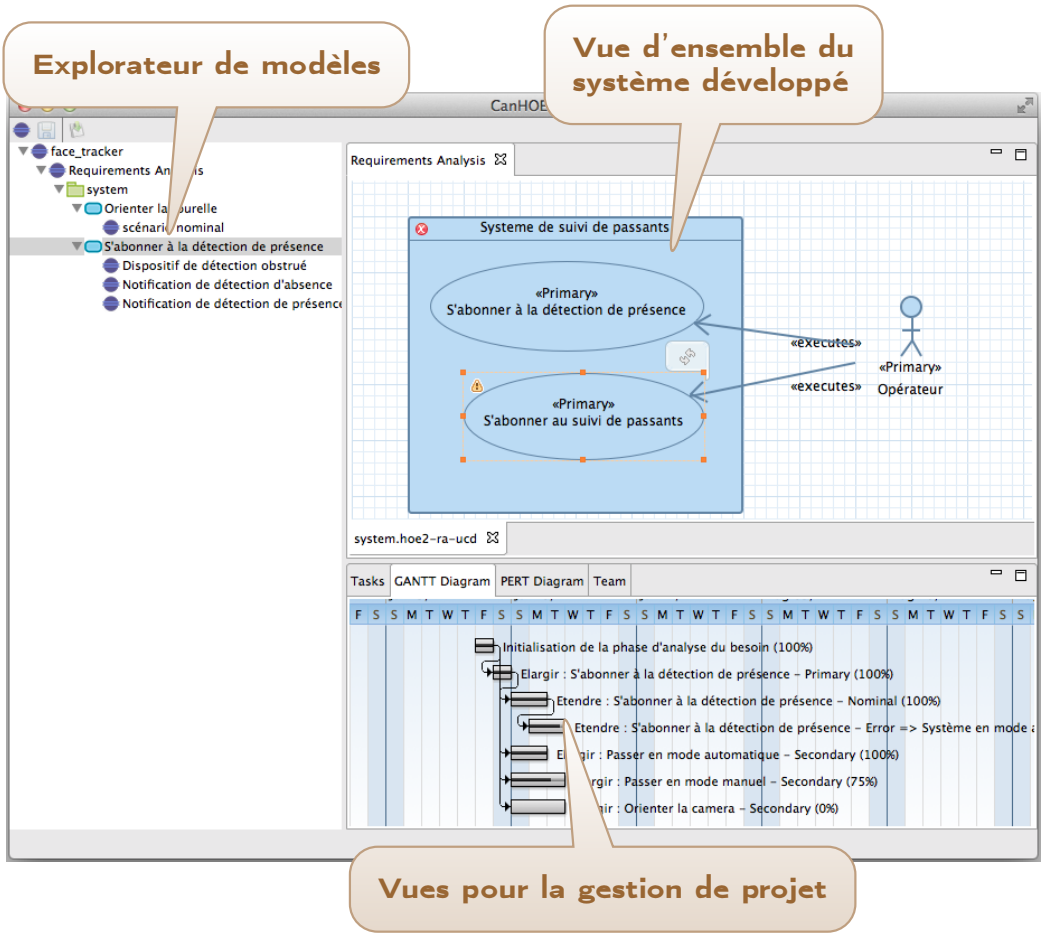


FIGURE 36 – Outillage CanHOE2 – perspective chef de projet

en lecture et d'intégration des différents modèles produits par les développeurs. Il ne possède en outre pas de palette d'outils lui permettant de réaliser des activités de modélisation. Ce choix a été fait afin de bien discerner l'activité de mesure et de pilotage du projet de celle de la conception réelle des modèles du système considéré. L'outil sera détaillé dans le chapitre 9.

Dans un souci d'assurer une gestion de projet efficace, des outils de gestions de projets (feuille de tâches, diagramme de Gantt) sont directement intégrés dans la vue du chef de projet et alimentés par l'état des modèles à tout instant. Le chef de projet peut ainsi mesurer l'avancement et assigner des tâches en interagissant directement avec l'outil.

Une autre originalité de l'outil réside en son système de dépôt de modèles. Tous les modèles sont archivés en différentes versions dans l'outil de gestion de versions Git. Les outils de gestion de versions sont généralement très prisés dans les différents domaines d'ingénierie pour leurs nombreux avantages, mais une mauvaise connaissance ou utilisation de ces outils peut rapidement gêner le développement plutôt que le faciliter, notamment dans des grandes équipes. De plus, dans le domaine des systèmes embarqués, ils sont généralement découplés des outils de modélisation, ce qui ne permet pas simplement de connaitre l'état d'avancement

d'un projet, mais aussi de le piloter. Afin de supporter efficacement l'ingénierie, nous avons fait le choix d'intégrer de façon transparente Git au sein de l'outil comme moteur de partage et d'intégration des différents modèles manipulés.

En résumé

- ✓ *Support de la collaboration par une gestion multi-utilisateurs pour l'édition de modèles*
- ✓ *Support efficace de l'ingénierie notamment par l'utilisation discrète de l'outil de gestion de versions décentralisée Git*
- ✓ *Outils de gestion de projet intégrée permettant au chef de projet de mesurer l'avancement du projet et de le piloter*

Synthèse du chapitre

Dans ce chapitre, nous avons présenté la méthode ⟨HOE⟩² dans son ensemble. La méthode ⟨HOE⟩² – pour *Highly Heterogeneous Object-Oriented Efficient Engineering* – propose un processus, un langage et un outillage dédié. Le processus proposé est itératif, guidé et aborde une approche descendante. Il est décliné en quatre phases permettant le développement de systèmes embarqués depuis la spécification du besoin jusqu'à son implémentation. Il permet de concevoir des modèles dans le langage dédié à ⟨HOE⟩². Ce langage étend le langage UML, en l'enrichissant de concepts propres au développement des systèmes embarqués. Enfin l'outillage dédié *CanHOE2* permet de guider le développeur et de s'assurer que le processus de développement est correctement suivi. Ce dernier contribue nettement à améliorer l'efficacité d'ingénierie et offre un support solide au chef de projet afin de mesurer et de piloter le développement.

Dans le chapitre suivant, nous détaillerons le processus canonique et collaboratif, ainsi que le langage produit de la méthode ⟨HOE⟩². Les aspects composition de plates-formes, gestion de projet et outil dédié seront respectivement présentés dans les chapitres 7, 8 et 9.

Chapitre 6

Description détaillée de la méthode ⟨HOE⟩²

Publications pertinentes à ce chapitre :

[**Hili *et al.* 2012b**] Nicolas Hili, Christian Fabre, Dominique Rieu, Sophie Dupuy-Chessa et Stéphane Malfoy. *⟨HOE⟩², une méthode intégrée pour le développement des systèmes embarqués.* In XXXème Congrès Inforsid'2012, Mai 2012

[**Hili *et al.* 2012a**] Nicolas Hili, Christian Fabre, Sophie Dupuy-Chessa et Stéphane Malfoy. *Efficient Embedded System Development: A Workbench for an Integrated Methodology.* In ERTS2 2012, Toulouse, France, Février 2012

Liste des sections

Dans le chapitre précédent, nous avons brièvement présenté les principaux aspects de la méthode ⟨HOE⟩². Cette méthode propose un processus rigide permettant un développement collaboratif de systèmes embarqués complexes, avec un langage commun pour les développeurs des applications et de la plate-forme du système.

Dans ce chapitre, nous détaillons le langage commun ainsi que le processus. nous n'aborderons ni les aspects liés à la gestion de projet, ni ceux liés à la composition de plates-formes. Ces deux aspects seront respectivement introduits dans les chapitres 8 et 7 qui leur seront consacrés. Par conséquent, le lecteur peut se référer à la figure 31 du chapitre précédent (page 76) pour visualiser le processus et les niveaux de modèle qui seront présentés dans ce chapitre.

Le chapitre est divisé en sept sections : deux premières sections d'introduction et cinq sections décrivant les quatre phases du processus ⟨HOE⟩² et le langage basé sur UML. La première section d'introduction permet de décrire l'organisation du chapitre. La seconde section permet de présenter le système qui nous servira de cas d'étude tout au long de ce chapitre, mais aussi tout au long des suivants.

Organisation du chapitre

Cette première section d'introduction permet d'expliquer l'organisation du chapitre afin d'en faciliter sa lecture. L'organisation du chapitre est guidée par les quatre phases du processus. Chaque section présentera une phase et sera scindée en deux parties : la première sera consacrée à la description du langage associé à la phase tandis que la seconde sera dédiée à la description de la démarche et de chacune des activités la composant. Le langage ne possède donc pas de section propre, il sera introduit graduellement, au fur et à mesure du processus.

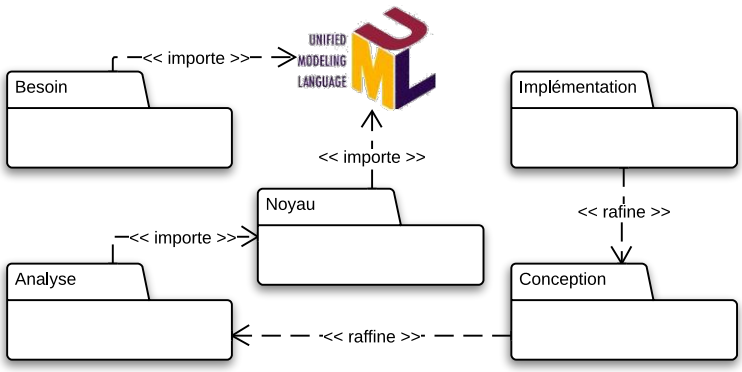


FIGURE 37 – Organisation du langage ⟨HOE⟩²

Présentation du langage. Le langage sera présenté par phase de processus. Dans la figure 30 présentée dans le chapitre précédent à la page 74, nous avions présenté le langage divisé selon les phases du processus. Concrètement, la figure 37 illustre l'organisation du langage. Le langage est composé de quatre méta-modèles pour les quatre phases du processus ⟨HOE⟩² et d'un méta-modèle commun, le Noyau, qui regroupe les éléments essentiels (objets, machines à états, etc.) pour la modélisation de systèmes. À l'exception du méta-modèle d'*analyse du besoin*, qui est spécifique, les trois autres se présentent comme des raffinements successifs. Enfin, dans la mesure où le langage ⟨HOE⟩² est proposé comme une extension du langage UML, les méta-modèles Noyau et Besoin importent un certain nombre de concepts UML.

La présentation du langage pour chacune des phases sera structurée de la manière suivante : (1) syntaxe abstraite (le méta-modèle) et ses contraintes ; (2) syntaxe concrète (la notation)

et exemples ; (3) règles de cohérence intra et inter-modèles. Les concepts du noyau seront introduits avant leur première utilisation, c'est-à-dire avant la description de la phase d'*analyse du système*. Le dictionnaire de l'ensemble des concepts est fourni en annexe A. Les contraintes du méta-modèle et les règles de cohérence sont exprimées en langage naturel et en Object Constraint Language (OCL). Les règles OCL sont données et expliquées pour la première phase du processus. Pour les autres phases, elles seront données dans l'annexe B.

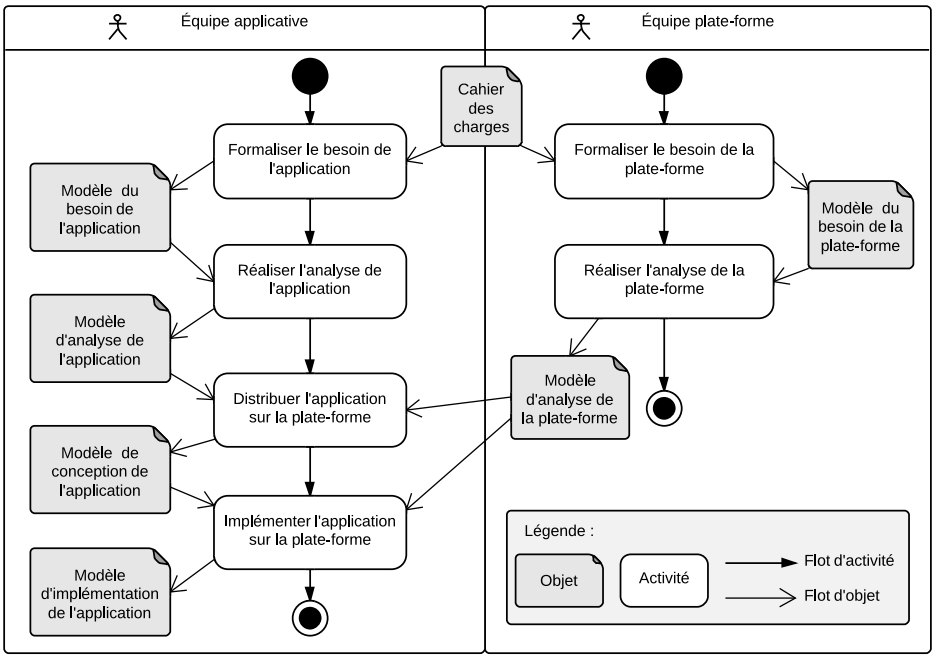
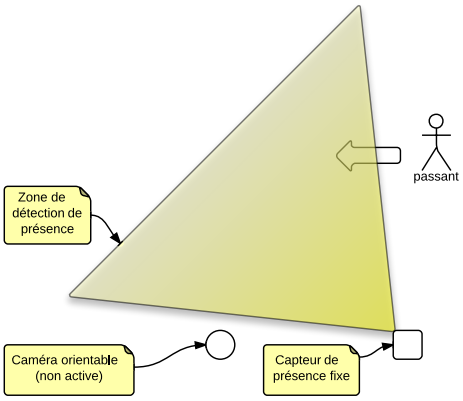


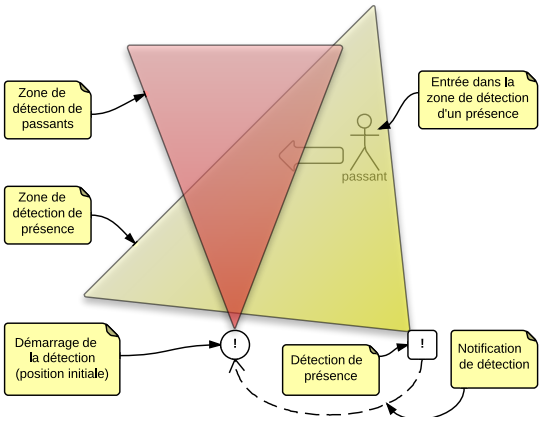
FIGURE 38 – Diagramme d'activités modélisant le processus

Présentation de la démarche. Tout au long de ce chapitre, le processus sera progressivement présenté par phase. La figure 38 illustre le processus de la figure 31 (cf. page 76) par un diagramme d'activités. Dans ce diagramme, les objets représentent les entrées et les sorties des activités. Le flot des activités est illustré par des flèches unidirectionnelles entre les activités. Ce diagramme illustre plusieurs originalités du processus : travail collaboratif entre les développeurs applicatifs et de la plate-forme ; des points de synchronisation permettant la collaboration ; des activités durant les premières phases du processus indépendante pouvant être réalisées en parallèle ; une introduction progressive de la plate-forme dans le flot de développement de l'application durant les phases de conception et de l'implémentation.

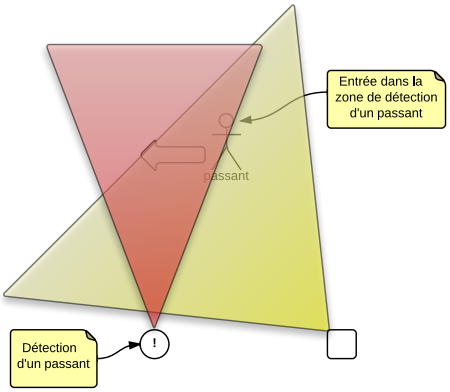
La présentation du processus pour chacune des phases sera structurée conformément au plan suivant : (1) description de la phase ; (2) illustration de la phase par un diagramme d'activités ; (3) description de chacune des tâches : tâche d'initialisation de la phase, de modélisation du système et de clôture de la phase.



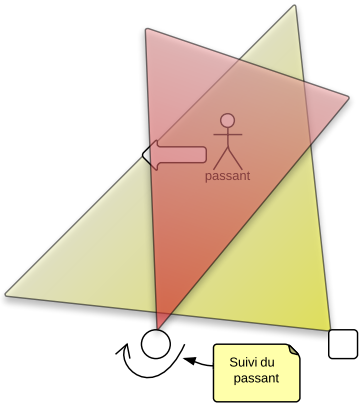
(a) position initiale



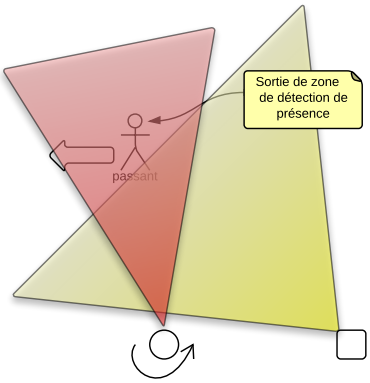
(b) Détection d'une présence



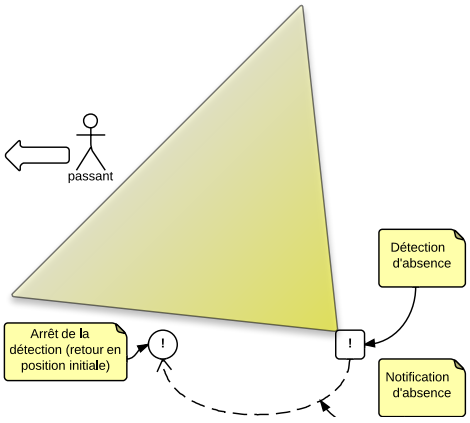
(c) Détection d'un passant



(d) Suivi du passant



(e) Suivi du passant en fin de zone



(f) Fin de détection

FIGURE 39 – Les différentes étapes de détection

Introduction au cas d'étude

Cette partie a pour objectif de présenter le cas d'étude qui sera utilisé tout le long de ce chapitre et des suivants. Il s'agit d'un système de suivi de passants[1]. Il permet, à l'aide d'une caméra orientable montée sur un support contrôlé par deux moteurs, de détecter et de suivre un passant dans une certaine zone. Ces deux moteurs permettent d'orienter la caméra selon deux axes. Un capteur de présence monté sur un bâti fixe, permet préalablement de détecter une présence. Ce système automatique peut également être basculé en mode manuel par un *opérateur* qui peut choisir de piloter manuellement la caméra. Cela peut être souhaité si cette dernière n'arrive pas à détecter correctement la position du passant, notamment en raison de conditions atmosphériques défavorables.

La figure 39 illustre les différentes étapes de détection du système : (a) initialement, seul le détecteur de présence fonctionne et couvre une large *zone de détection* ; (b) lorsque qu'un passant entre dans la zone de détection de présence, le capteur notifie la caméra qui s'active alors et couvre une *zone de détection de visage* depuis sa position initiale ; (c) lorsque le passant entre dans la zone de détection de visage, la caméra le détecte et (d) commence à pivoter afin de centrer l'objet dans la zone de détection de visage. (e) le passant continue son déplacement et la caméra le suit toujours jusqu'à sa sortie de la zone de détection de présence. (f) Lorsque ce dernier a totalement quitté la zone de détection de présence, le capteur de présence en notifie la caméra qui se replace alors en position initiale et arrête la détection de visage.

1 Méta-modèle et phase d'analyse du besoin

La phase d'analyse du besoin permet de formaliser un besoin informellement décrit à partir d'un *cahier des charges*, sous la forme d'un modèle. Ce modèle représente le *système* à développer, l'ensemble des *cas d'utilisation* qu'il doit satisfaire et l'énumération des différents *acteurs* qui utilisent le système. Un ensemble de *scénarios* pour chaque cas d'utilisation décrit comment le système se comporte à partir d'un stimulus provenant d'un acteur du système. L'activité de formalisation du besoin s'effectue au moyen de modèles UML enrichis. En plus d'ajouts sur le langage qui seront détaillés plus loin, l'une des spécificités du processus ⟨HOE⟩[2] dès la première phase réside dans la séparation des cas d'utilisation relevant de l'application de ceux de la plate-forme. Cette séparation permet de distinguer des cas d'utilisation par essence fondamentalement différents et de paralléliser très tôt le processus. En effet, les cas d'utilisation de l'application relèvent principalement du domaine ciblé, tandis que les cas d'utilisation de la plate-forme relèvent essentiellement de la gestion des applications installées.

1.1 Le langage

Syntaxe abstraite. La figure 40 illustre le méta-modèle du besoin étendant le méta-modèle UML. Les classes blanches représentent notre extension. Le méta-modèle du besoin est proposé au sein d'un méta-modèle nommé HOE2::Besoin. Les méta-classes en rose représentent une portion du méta-modèle UML. Seule la partie permettant de comprendre la façon dont nous avons étendu et spécialisé le méta-modèle a été représentée. Les méta-classes d'UML utilisées appartiennent à sept différents paquetages UML définis dans le tableau 15.

1. Dans la suite du chapitre, il pourra plus simplement être appelé « *système* »

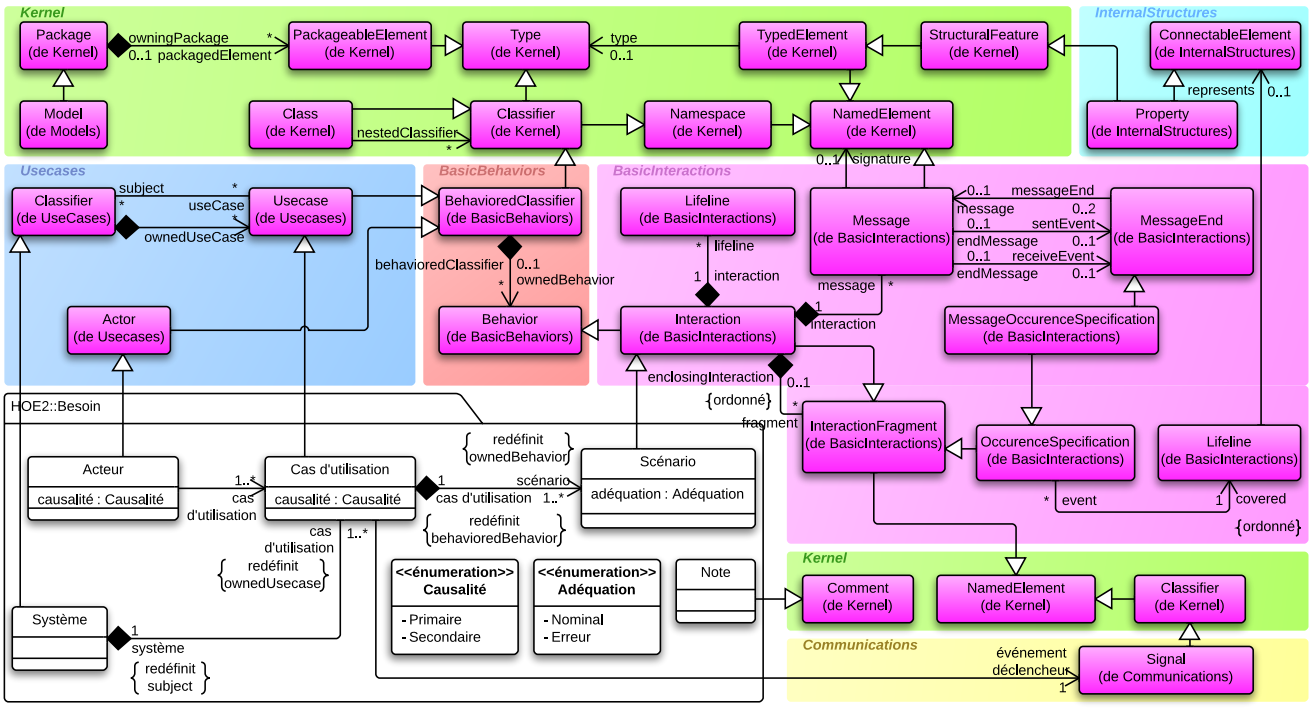


FIGURE 40 – Méta-modèle d'analyse du besoin

Paquetage	Description
BasicBehaviors	Contient un ensemble de méta-classes permettant de décrire de manière comportementale les différentes connexions entre objets.
BasicInteractions	Contient les méta-classes pour la modélisation d'intéractions.
Communications	Contient les méta-classes nécessaires pour la modélisation des communication et l'invocation des comportements.
Kernel	Le noyau commun d'UML contenant les méta-classes pouvant être utilisées dans tous les autres packages.
Models	Paquetage raffinant le paquetage Kernel. Il fournit l'unique méta-classe *Model*
InternalStructures	Contient les méta-classes permettant de décrire structurellement les différentes connexions entre objets.
Usecases	Contient les méta-classes nécessaires pour la modélisation de cas d'utilisation.

TABLE 15

Le méta-modèle d'analyse du besoin capture tous les éléments permettant de modéliser le besoin du *système considéré* de manière formelle. Les fonctionnalités du *système considéré* sont

capturées à l'aide de *cas d'utilisation* et de *scénarios*. Par rapport aux concepts de UML, les acteurs et cas d'utilisation du langage ⟨HOE⟩[2] possèdent une *causalité* par rapport au *système*. Deux valeurs sont possibles : un cas d'utilisation ou un acteur est *primaire* s'il est la raison pour laquelle le système est conçu. Il est *secondaire* s'il est une conséquence du développement du système. Les scénarios sont également rangés dans deux catégories : les scénarios *nominaux*, décrivant les comportements normaux ou attendus du système lors du déclenchement d'un cas d'utilisation par un acteur, et les scénarios d'*erreur* qui décrivent des comportements anormaux ou inattendus du système. Des *notes* permettent d'annoter les différents éléments du modèles.

Id	Description

[1]	Au moins un acteur primaire doit être présent dans le modèle.
	`context Model inv:`
	`  self.packagedElement->selectByType(Acteur)->exists(a : Acteur | a.`
	`  causalité = Causalité::Primaire)`

[2]	Au moins un cas d'utilisation primaire doit décrire le système.
	`context Système inv:`
	`  self.cas d'utilisation->exists(cu : Cas d'utilisation | cu.causalité =`
	`  Causalité::Primaire)`

[3]	Au moins un scénario nominal doit décrire chaque cas d'utilisation.
	`context Cas d'utilisation inv:`
	`  self.scénario->exists(sc : Scénario | sc.nature = Nature::Nominal)`

[4]	Un acteur secondaire ne peut déclencher que des cas d'utilisation secondaires.
	`context Acteur inv:`
	`  self.causalité = Causalité::secondaire implies self.cas d'utilisation`
	`  ->forAll(cu : Cas d'utilisation | cu.causalité = Causalité::`
	`  secondaire)`

[5]	Un scénario commence toujours par un message de l'acteur vers le cas d'utilisation.
	`context Scénario inv:`
	`  self.fragment->selectByType(OccurrenceSpecification)->first().covered.`
	`  represents.oclAsType(Property).type.oclIsTypeOf(Acteur)`

TABLE 16 – Liste des contraintes du méta-modèle d'analyse du besoin

Les trois premières contraintes sont liées aux cardinalités des concepts (acteur primaire, cas d'utilisation primaire et scénario nominal) dans le méta-modèle. La première contrainte s'applique directement sur la méta-classe *Model* du méta-modèle UML. L'opération `collection->exists(v : Type | expr)` vérifie qu'au moins un élément v d'une collection vérifie la condition *expr*. `collection->selectByType(type : Classifier) : Collection(T)` retourne une sous-collection contenant tous les éléments dont le type correspond à *type*.

La quatrième contrainte interdit l'association d'un acteur secondaire à un cas d'utilisation primaire (mais n'empêche pas un cas d'utilisation secondaire d'être déclenché par un acteur primaire). L'usage du mot-clé `implies` permet de vérifier l'égalité entre deux expressions booléennes, tandis que l'opération `collection->forAll(v : Type | expr)` vérifie que tous les éléments v d'une collection valide la condition *expr*. Enfin, la dernière contrainte vérifie la bonne construction de l'ensemble des scénarios qui doit toujours commencer par un message provenant de l'acteur. Ici, l'opération `collection->first()` permet de sélectionner le premier élément d'une

suite ordonnée. Les opérations `oclAsType(type : Classifier) : T` et `oclIsTypeOf (type : Classifier ) : Boolean` permettent respectivement de convertir ou de vérifier le type d'un élément.

Exemples et notations. La notation graphique est grandement inspirée d'UML avec quelques variations. Deux diagrammes sont disponibles pour concevoir les modèles durant cette phase : le diagramme d'*analyse du besoin* étendant le diagramme de cas d'utilisation d'UML, et le diagramme de *scénario* étendant celui de séquences.

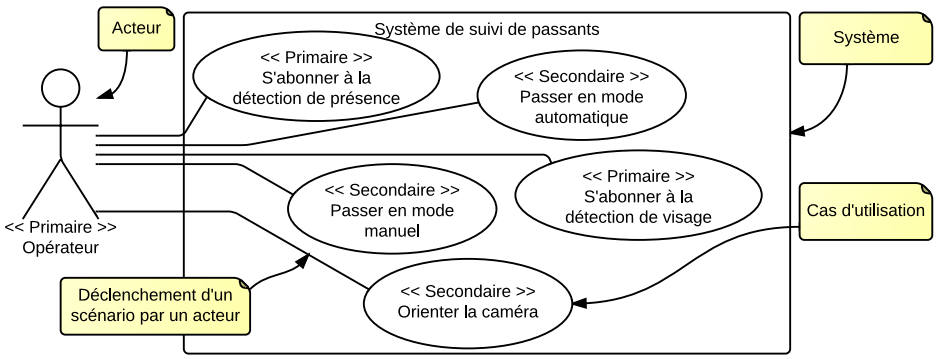
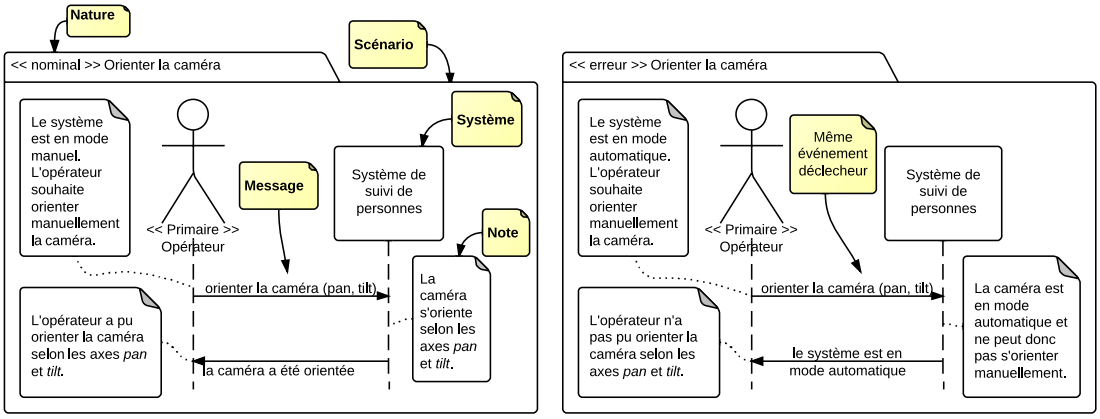


FIGURE 41 – Diagramme d'analyse du besoin de l'application

Le diagramme d'*analyse du besoin* de l'application présenté précédemment est illustré par la figure 41. Dans ce diagramme, nous avons modélisé l'acteur *opérateur* qui peut choisir de basculer le système en mode automatique ou en mode manuel. En mode automatique, il se contente de recevoir des données provenant du système, telles que la détection de présence, la détection de visage et dans la mesure du possible, la position du visage. En mode manuel, il peut en plus orienter lui-même la caméra. La notation graphique est la même que les diagrammes de cas d'utilisation d'UML avec pour seule nuance l'affichage – sous la forme de stéréotypes UML – de la causalité des acteurs et des cas d'utilisation au dessus de leurs noms respectifs.



(a) Scénario nominal (b) Scénario d'erreur

FIGURE 42 – Diagrammes de scénario du cas d'utilisation « Orienter la caméra »

Pour chaque cas d'utilisation, un ensemble de scénarios est défini. Chaque scénario est modélisé graphiquement grâce à un diagramme de *scénario* tel qu'illustré par la figure 42. Les figures 42a et 42b illustrent deux scénarios du même cas d'utilisation, permettant d'orienter la caméra. Le premier scénario est le scénario nominal du cas d'utilisation, lorsque tout se passe bien, tandis que le second modélise un cas d'erreur. Des notes permettent de décrire le scénario. Une première note indique dans quel état se trouve le système et décrit l'action de l'acteur externe. Une dernière note permet de décrire l'état du système ou de l'acteur à la fin du déclenchement du scénario. Des notes intermédiaires décrivent le comportement du système ou de l'acteur à la réception d'un message. Des règles de cohérence permettent d'assurer la bonne construction des scénarios par rapport au besoin modélisé dans le diagramme *d'analyse du besoin*. Par exemple, tous les scénarios d'un même cas d'utilisation commencent par le même événement déclencheur. Toutefois, l'état initial du système peut différer, justifiant plusieurs scénarios (nominaux ou d'erreur) pour un même cas d'utilisation.

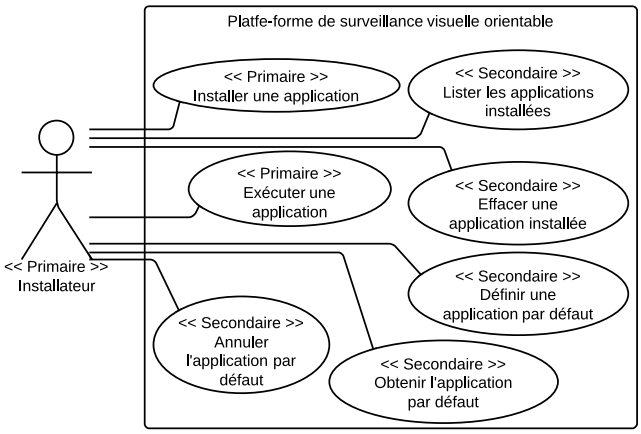


FIGURE 43 – Diagramme d'analyse du besoin de la plate-forme

De la même manière, il est possible de définir les cas d'utilisation de la plate-forme hébergeant l'application. La figure 43 illustre celui de la plate-forme de contrôle de la caméra PT [2]. Cette dernière permet essentiellement de gérer le cycle de vie (installation, exécution, arrêt, désinstallation) des applications qu'elle est capable d'héberger. Elle offre en outre des cas d'utilisation supplémentaires pour définir et obtenir l'application par défaut de la plate-forme.

Id	Description
[1]	Tous les scénarios d'un même cas d'utilisation commencent par le même événement déclenchant, défini par le cas d'utilisation.

```
context Cas d'utilisation inv:
  self.scénario->forAll(s : Scénario | s.fragment->selectByType(
    MessageOccurenceSpecification)->first().message.signature = self.
    événement déclencheur)
```

TABLE 17 – Règle de cohérence intra-model du besoin

2. PT désigne *pan* & *tilt*, les deux angles conventionnels d'orientation d'une caméra.

Règles de cohérence intra-modèle.

Le tableau 17 résume la règle de cohérence intra-modèle de la phase d'analyse du besoin. Elles s'appuie sur les relations entre le système modélisé dans le diagramme d'*analyse du besoin* et les diagrammes de *scénarios*. Une seule règle a été définie afin d'assurer que tous les scénarios démarrent par le même message, l'événement déclencheur du cas d'utilisation.

1.2 La démarche

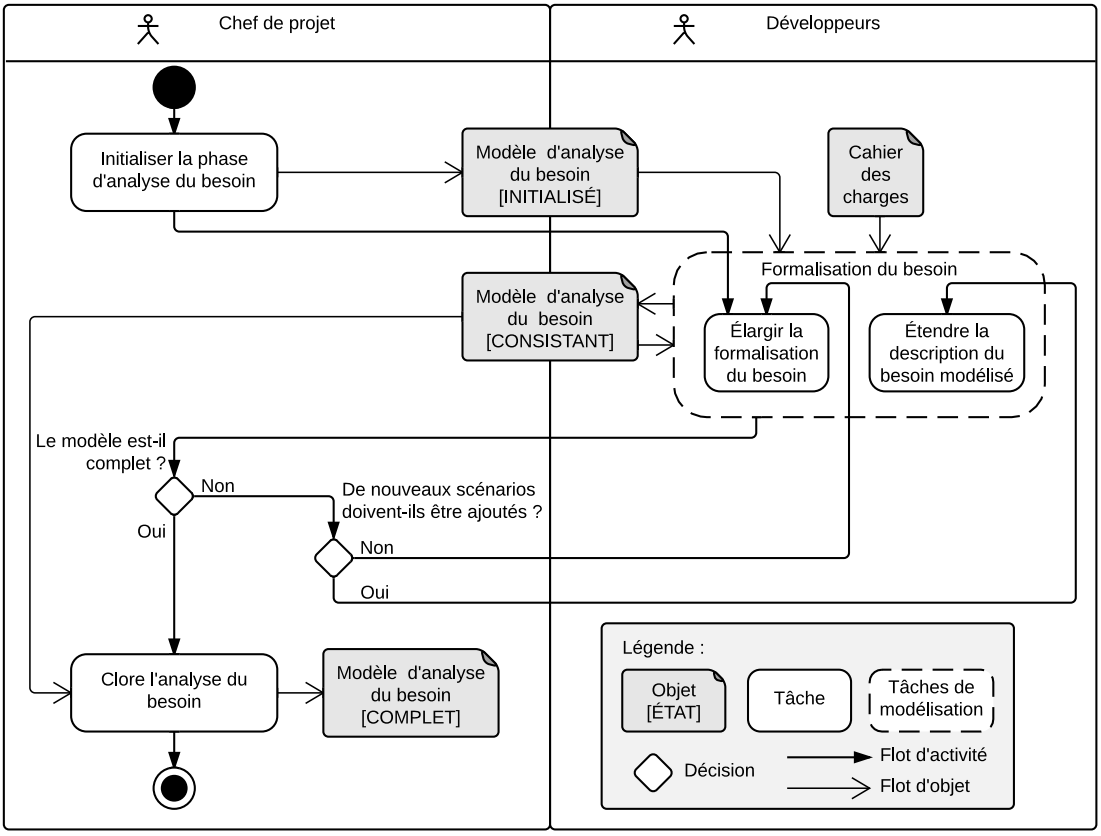


FIGURE 44 – Démarche de l'analyse du besoin

La figure 44 illustre les différentes tâches durant la phase d'*analyse du besoin* à l'aide d'un diagramme d'activités. Dans cette phase ainsi que dans les suivantes, nous illustrons les tâches réalisées par le chef de projet qui résident essentiellement dans l'initialisation et la clôture des phases. La réalisation des tâches de modélisation revient aux développeurs du système. Nous avons utilisé la notation UML pour la représentation des différentes tâches de chacune des phases. La notation de la méta-classe UML *Action* – un rectangle aux bords arrondis – est utilisée pour représenter les tâches du processus. Pour la tâche de modélisation de chacune des phases, qui consiste généralement en plusieurs tâches, nous avons choisi la notation du *nœud structuré d'activités* (méta-classe UML *StructuredActivityNode*). Les décisions sont prises par le chef de projet qui initialise la phase et assige aux développeurs les tâches de modélisation du

modèle d'analyse du besoin. Le cahier des charges est une entrée du processus à partir duquel le besoin est formalisé.

Initialisation de la phase. La phase d'analyse du besoin est la première phase du processus. À ce titre, la première tâche consiste à définir le nom du système à développer. Le système peut être indépendamment une application ou une plate-forme d'un système embarqué. Cette première tâche initialise le modèle d'*analyse du besoin* qui se retrouve alors dans l'état INITIALISÉ. Dès son initialisation, le modèle contient le système à développer.

Tâches de modélisation « *Formalisation du besoin* ». La tâche de *formalisation du besoin* consiste à couvrir tous les besoins spécifiés dans le cahier des charges. Lors de la première itération, le développeur *élargit* la formalisation du besoin pour produire un modèle du besoin CONSISTANT. Ce modèle contient au moins un système, un acteur primaire qui exécute un cas d'utilisation primaire, ce dernier étant défini par au moins un scénario nominal. Une fois cette première tâche réalisée, le développeur peut *étendre* la description actuelle du besoin modélisé en ajoutant des scénarios aux cas d'utilisation modélisés ou continuer d'*élargir* la formalisation en ajoutant de nouveaux cas d'utilisation et acteurs. Lorsque tout le besoin du cahier des charges a été formalisé, le chef de projet peut *clore* la phase.

Clôture de la phase. La tâche de clôture est la dernière réalisée sur le modèle du besoin. Elle est à l'initiative du chef de projet qui considère que l'ensemble du besoin du cahier des charges nécessitant d'être formalisé a été modélisé dans le modèle d'analyse du besoin. Lorsque le chef de projet réalise cette activité, le modèle d'analyse du besoin est alors COMPLET. La phase est close et aucune nouvelle activité de modélisation ne peut être réalisée.

La démarche présentée pour la première phase du processus présente plusieurs avantages. Du point de vue collaboratif d'abord, avec la présence du chef de projet pour les prises de décision et la réalisation des tâches d'initialisation et de clôture de la phase, ainsi que le regroupement des tâches de modélisation du côté du développeur. Du point de vue de l'efficacité de l'ingénierie et de la rapidité du développement ensuite, la présence d'un état intermédiaire CONSISTANT du modèle permet de démarrer en avance les autres phases du processus et ainsi accélérer le développement de prototypes avant que tout le besoin ne soit formalisé.

En résumé

- ✓ *Méta-modèle d'analyse du besoin spécialisant le méta-modèle UML*
- ✓ *Deux diagrammes étendant les diagrammes de cas d'utilisation et de séquences d'UML*
- ✓ *Réutilisation de la notation UML avec quelques variations*
- ✓ *Mêmes tâches de modélisation pour une application ou une plate-forme*
- ✓ *État* CONSISTENT *du modèle permettant de démarrer très tôt l'analyse du système*
- ✓ *Les cas d'utilisation de la plate-forme adressent la gestion des applications qu'elle héberge*

2 Méta-modèle Noyau

Avant d'entamer la description des trois phases suivantes, il est nécessaire de présenter les concepts communs organisés au sein du méta-modèle *Noyau* (cf. fig. 37 de la page 86). Cette section présente le méta-modèle *Noyau*, dont les concepts seront utilisés dans les phases suivantes.

2.1 Description du méta-modèle Noyau

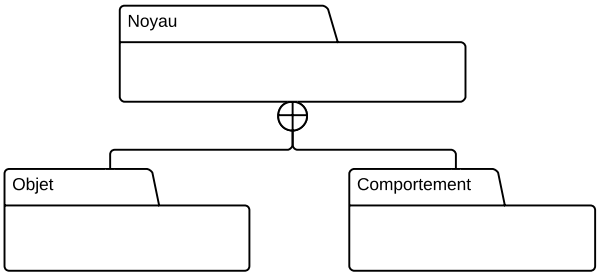


FIGURE 45 – Composition du méta-modèle Noyau

La figure 45 illustre l'organisation du méta-modèle Noyau. Ce dernier ne contient pas d'éléments mais est composé de deux sous-paquetages, respectivement Objet et Comportement. Le paquetage Objet contient les concepts basiques permettant de modéliser les objets composant le système embarqué dans ⟨HOE⟩². Le paquetage Comportement contient les concepts pour modéliser les aspects comportementaux dans ⟨HOE⟩². Ces deux paquetages sont décrits dans cette section avant d'entamer la présentation de la phase d'*analyse du système*.

2.2 Le paquetage objet

Syntaxe abstraite. La figure 46 illustre le méta-modèle des objets ⟨HOE⟩², étendant et spécialisant le méta-modèle UML. Les méta-classes d'UML présentent un fragment partiel suffisant pour la compréhension de notre extension. UML permet de construire des modèles à partir de composants interconnectés. Ces composants peuvent dialoguer de manière synchrone (opérations) ou asynchrone (signaux). Nous privilégions le seul envoi asynchrone par envoi de signaux (que nous renommerons pour la suite *messages*) représentatif du comportement réel de la communication entre composants d'un système embarqué.

Le concept principal dans le langage ⟨HOE⟩² est celui de l'*objet*. Il est constitué d'un ensemble de messages qu'il peut émettre vers ou recevoir de la part d'autres objets par l'intermédiaire d'*associations*. L'ensemble des messages qu'il peut émettre ou recevoir est l'union de l'ensemble des messages de ses interfaces *requises* et *fournies*, référencées par la méta-classe d'UML *Component* que nous avons décidé d'étendre. Dans ⟨HOE⟩², un *acteur* est lui-même un objet, externe au système. Il ne possède pas d'états, mais contient également un ensemble de messages qu'il peut envoyer et recevoir. En proposant le concept unique d'objet, à la fois pour le développement applicatif et de la plate-forme, nous offrons un langage commun et diminuons le nombre de concepts à manipuler ainsi que l'effort d'apprentissage. L'inclusion de

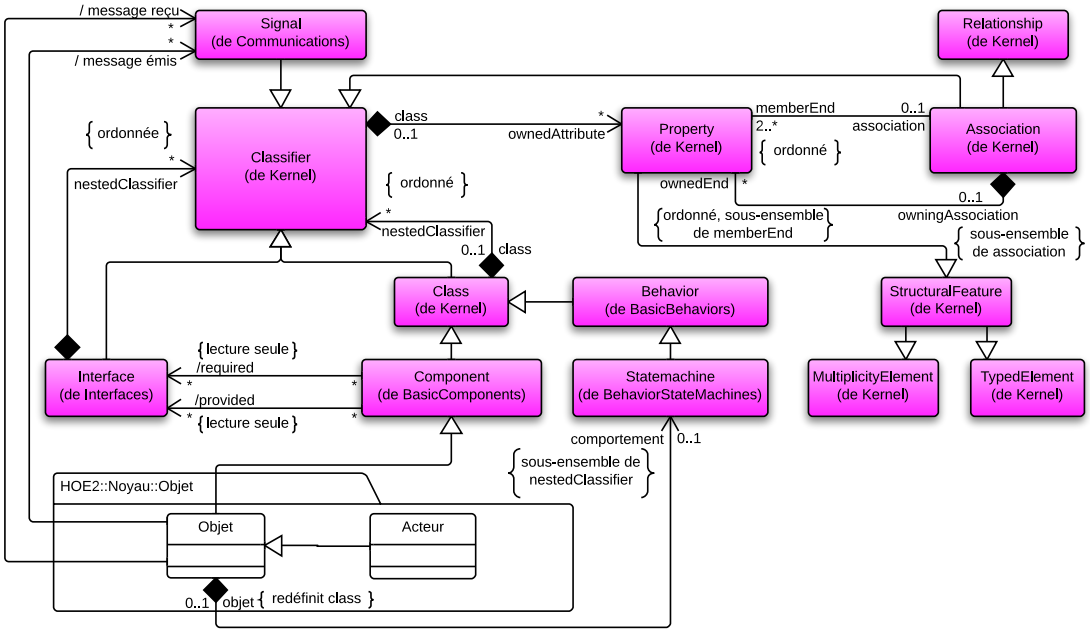


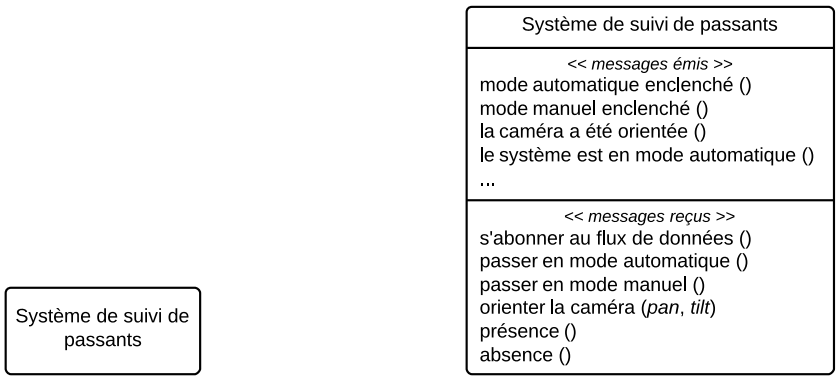
FIGURE 46 – Méta-modèle des objets ⟨HOE⟩^{2}

l'acteur comme concept commun nous permet également de pouvoir l'utiliser à tous les niveaux d'abstraction. Ainsi, il est facile de conserver la cohérence avec la phase d'analyse du besoin.

Un objet peut communiquer avec d'autres objets au travers d'*associations*. Nous avons conservé la sémantique des associations UML. L'objet à l'extrémité d'une association est identifiée par son nom de *rôle* et sa *cardinalité*. UML offre une très grande expressivité (agrégations, compositions, associations n-aires, classes d'association, etc.). Dans ⟨HOE⟩^{2}, les associations permettent de modéliser les connexions entre les objets pouvant dialoguer par échange de messages. Pour cette raison, nous avons limité l'utilisation de l'association, en interdisant les différents types d'association mentionnées plus haut [3]. Tout comme UML, où les associations sont considérées comme des composants à part entière – le concept *Association* de UML étend celui de *Classifier* –, dans ⟨HOE⟩^{2}, nous considérons les associations comme des objets. À ce titre, il est possible de modéliser leur comportement – i.e. leur logique de communication – par une machine à états. Les réseaux sur puces NoC [4] sont des exemples de systèmes dans lesquels il est pertinent d'implémenter une logique d'acheminement des messages.

Exemple et notations. La figure 47 illustre la modélisation du premier objet du *système de suivi de passants*. La figure 47 fournit deux représentations graphiques de l'objet : une représentation simplifiée (à gauche) où seul le nom de l'objet apparait et une représentation détaillée (à droite) avec la liste des messages de l'objet. Nous avons fait le choix d'une représentation graphique similaire à la représentation graphique UML des classes afin de ne pas alourdir l'effort d'apprentissage, avec toutefois une différence : les interfaces ne sont pas affichées. À la

3. Les agrégations jouent un rôle particulier dans le processus ⟨HOE⟩^{2}, comme nous le verrons plus loin
4. En anglais, Network-on-Chip (NoC).



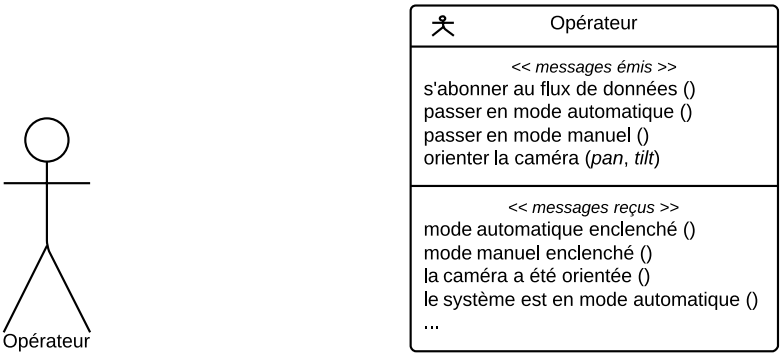
(a) Représentation simple (b) Représentation détaillée

FIGURE 47 – Représentations graphique d'un objet ⟨HOE⟩²

place, les messages requis ou fournis (respectivement) de chacune des interfaces de l'objet sont respectivement affichés dans le second et le troisième compartiment de l'objet (sur la figure 47b).

La figure 48 fournit deux représentations graphiques de l'acteur dans ⟨HOE⟩² : une représentation simplifiée (à gauche), provenant de la notation graphique UML, où à nouveau seul le nom de l'acteur apparait et une représentation détaillée (à droite), similaire à la notation graphique de l'objet ⟨HOE⟩² (cf. fig. 47b) à laquelle est ajoutée une icône pour différencier l'acteur d'un objet classique.

La figure 49 illustre l'association entre trois objets. L'acteur Opérateur, le Système et un objet Détecteur de présence. Ce dernier est en charge d'alerter le système en cas d'une présence ou d'une absence détectée. Deux associations sont modélisées dans la figure 49. La première part de l'Opérateur et cible le Système. Le système est identifié par son nom de rôle système et par son unicité (cardinalité = 1). La seconde association part du Système et cible le Détecteur de présence. Ce dernier est identifié par son nom de rôle dp et son unicité (cardinalité = 1). On peut voir sur la figure que les messages émis par des objets sont des messages reçus par



(a) Représentation simple (b) Représentation détaillée

FIGURE 48 – Représentations graphique d'un acteur ⟨HOE⟩²

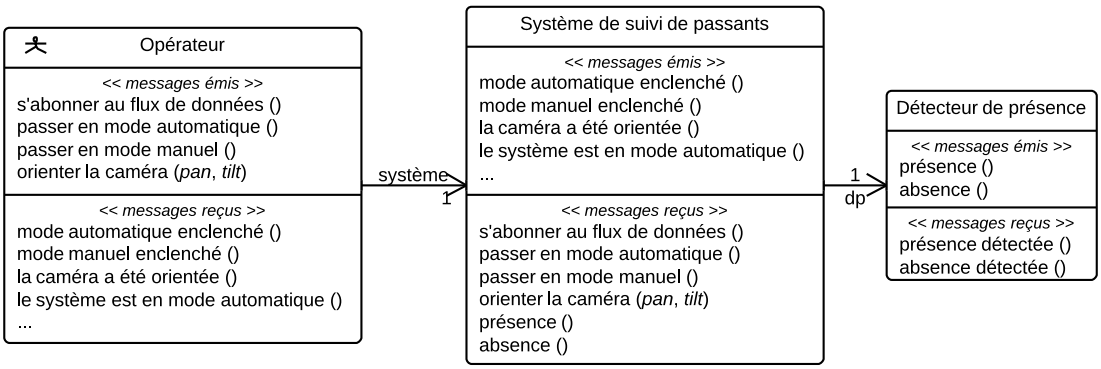


FIGURE 49 – Association entre plusieurs objets

d'autres, qui doivent alors être accessibles au travers d'une association.

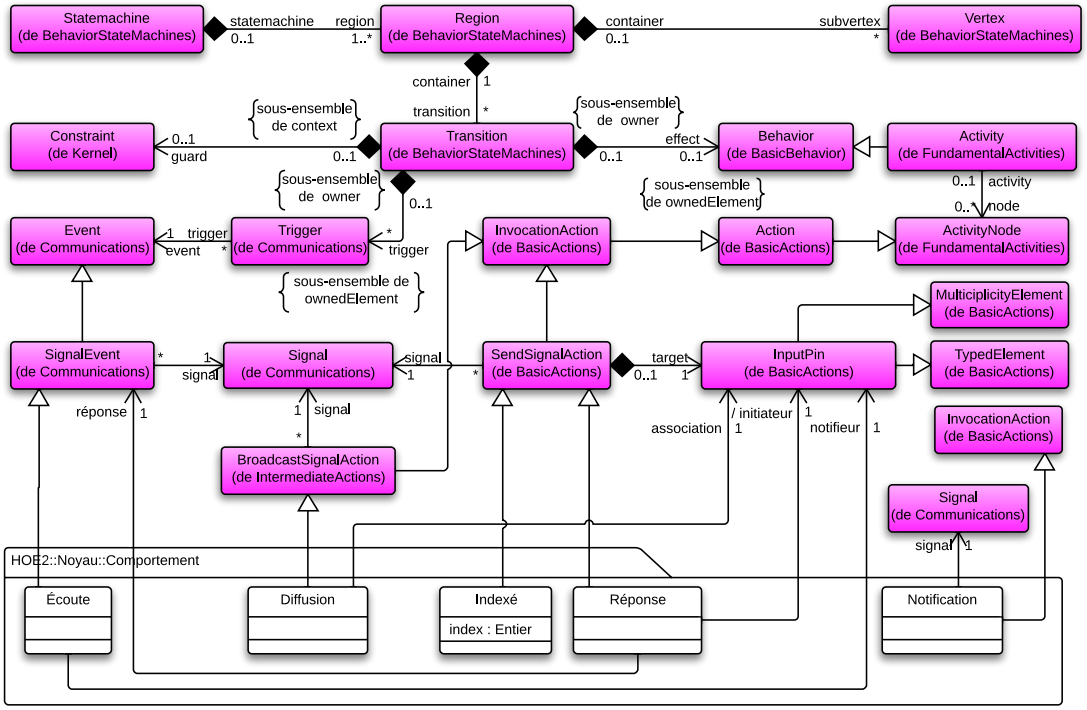


FIGURE 50 – Méta-modèle de comportement ⟨HOE⟩[2]

2.3 Le paquetage Comportement

Le *comportement* de l'objet est défini par une *machine à états*. Les concepts que nous avons étendus concernent exclusivement la *transition*. Les autres concepts (machine à états, régions, états initiaux, finaux, etc.) restent inchangés. Les travaux sur les transitions ont été réalisés en

complémentarité des travaux de thèse de Llopard et présentés dans [Llopard *et al.* 2014]. Dans cette partie, nous présentons le méta-modèle *de comportement* ⟨HOE⟩² étendant UML.

Syntaxe abstraite. Le méta-modèle de *comportement* ⟨HOE⟩² est illustré par la figure 50. Une transition traditionnellement définie dans les machines à états UML, illustrée par la figure 51, est constituée de trois parties : *événement*, *garde* et *effet* (ou *action*).

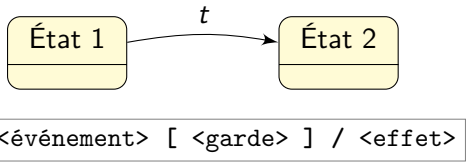


FIGURE 51 – Transition des machines à états UML

Dans ces travaux, nous avons décidé de conserver la définition des gardes UML et d'étendre les définitions d'*événement* et d'*effet* de la transition UML en limitant leurs portées et en les spécialisant. En limitant tout d'abord, puisque nous ne considérons que les *envois* et *réceptions* de *message* comme *effets* et *événements* des transitions. En spécialisant ensuite, puisque nous avons identifié les différentes réceptions et émissions qui nous semblent pertinentes afin de modéliser le comportement des systèmes embarqués. Ainsi, dans la figure 50, nous proposons cinq nouveaux concepts, un pour la *réception* de messages et quatre pour l'*émission* de messages, et avons identifié deux types de réception et cinq types d'émission de message dont les classifications sont données dans les tableaux 18 et 19.

Exemple et notations. Cette partie présente notre notation graphique pour la modélisation des comportements dans ⟨HOE⟩². Les tableaux 18 et 19 résument la liste des émissions et des réceptions de messages que nous proposons.

Le tableau 18 récapitule les différents types d'envoi de messages qu'il est possible de réaliser. La première colonne identifie l'association sur laquelle le type d'envoi de message est possible. La seconde colonne illustre l'émission de messages dans la machine à états de l'objet qui émet le message. Dans les quatre premiers cas, il s'agit du comportement d'un objet A, tandis que dans le dernier, il s'agit de celui d'un objet B. La troisième colonne illustre un fragment du méta-modèle permettant de modéliser le type d'émission. Les cinq types sont : *simple* (envoi d'un message à un objet unique), *réponse* (envoi d'un message comme réponse à un autre message), *indexé* (envoi d'un message à un objet identifié par son index dans un ensemble d'objets), *diffusion* (envoi d'un message à tous les objets de l'ensemble) et enfin, *notification* (envoi d'un message à tous les *écouteurs* d'un message – les écouteurs s'enregistrent avec le mot-clé **écouter** qui sera présenté dans le tableau 19).

Dans ⟨HOE⟩², nous autorisons les messages sous forme de séquences ordonnées ou bien en parallèle. Pour cela, sur une même transition, il est possible d'indiquer plusieurs messages, séparés par un point virgule (;) pour indiquer une séquentialité de deux messages (le message *m1* est envoyé, puis c'est au tour du message *m2*), ou séparés par une virgule (,) pour indiquer deux messages envoyés simultanément (les messages *m1* et *m2* sont envoyés en même temps).

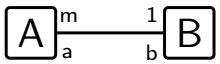
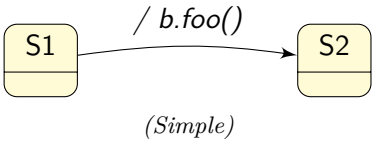
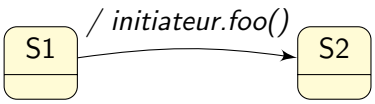
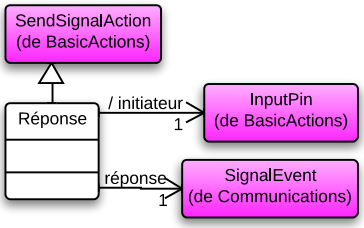
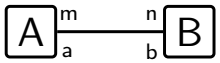
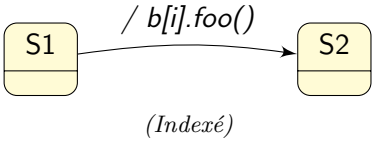
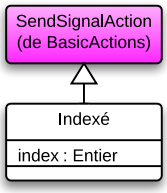
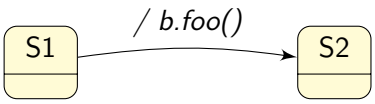
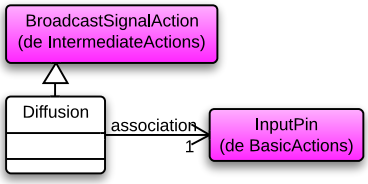
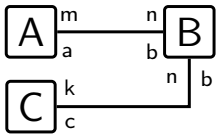
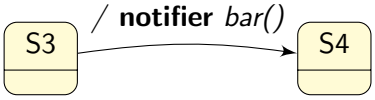
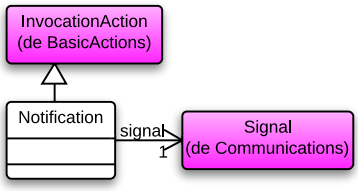
Association	Émission (comportement de A)	Fragment des méta-modèles ⟨HOE⟩²/ UML utilisés
	 *(Simple)*	
	 *(Réponse)*	
	 *(Indexé)*	
	 *(Diffusion)*	
Association	**Émission (comportement de B)**	**Méta-classe UML étendue**
	 *(Notification)*	

TABLE 18 – envoi de messages dans ⟨HOE⟩²

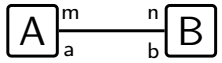
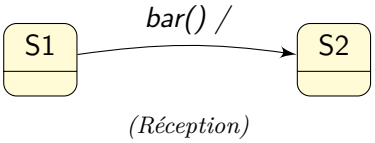
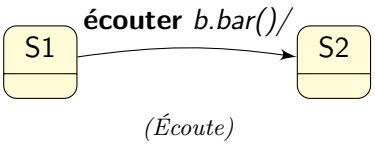
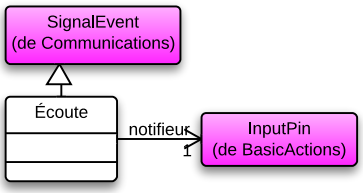
Association	Réception (comportement de A)	Méta-classe UML étendue
		
		

TABLE 19 – Réception de messages dans ⟨HOE⟩²

Le tableau 19 illustre les différentes *réceptions* autorisées dans ⟨HOE⟩². La *réception* classique (un message attendu est reçu) est basée sur la méta-classe UML *SignalEvent*. Elle permet de déclencher le franchissement de la transition dans le cas où le message est reçu. Une garde peut également être définie et dans ce cas évaluée avant le franchissement de la transition. La seconde est l'*écoute* (un objet s'inscrit en tant qu'*écouteur* d'un objet identifié et peut de ce fait recevoir les *notifications* de cet objet lorsque ce dernier utilise le mot-clé **notifier**).

Id	Description
[1]	Tous les messages intervenant dans un *envoi* de message d'un objet sont forcément des message *émis* par l'objet, c'est-à-dire contenus par l'une de ses interfaces *fournies*.
[2]	Tous les messages intervenant dans une *réception* de message d'un objet sont forcément des messages *reçus* par l'objet, c'est-à-dire contenus par l'une de ses interfaces *requises*.
[3]	Un envoi de message d'un objet *expéditeur* vers un objet *destinataire* n'est valide que si le *destinataire* est directement accessible par l'*expéditeur* au travers d'une association.
[4]	Une *diffusion* de message ou un envoi de message *indexé* implique que le message est transmis sur une association *multi-valuée* et *ordonnée*.
[5]	Si un objet (l'*écouteur*) *écoute* le message émis par un second objet (le *notifieur*), alors le *notifieur* doit être accessible au travers d'une association et le message doit appartenir à une interface fournie par le *notifieur* et requise par l'*écouteur*.

TABLE 20 – Liste des règles de cohérence intra-modèle du noyau

Règles de cohérence intra-modèle. Cette partie fournit quelques règles de cohérence pour la construction des modèles bâtis sur les concepts d'objet, de machine à états et d'association dans ⟨HOE⟩². Elles sont répertoriées dans le tableau 20. Ces règles permettent d'assurer que la machine à états d'un objet est bien construite et est conforme à la structure de l'objet et de ses

associations avec les autres objets. En particulier, les deux premières assurent la cohérence entre les messages reçus et émis dans une transition d'une machine à états et ceux contenus dans les interfaces *fournies* et *requises* des objets. Les deux règles suivantes assurent la présence d'une association entre les objets *expéditeur* et *destinataire* avec une contrainte supplémentaire pour l'envoi de message *indexé*. La dernière assure qu'un *notifieur* est bien accessible par l'*écouteur*. Ainsi, l'*écouteur* peut s'enregistrer auprès du *notifieur* avec le mot-clé **écouter**.

En résumé

- ✓ *Un méta-modèle commun pour la modélisation d'objets, d'associations et de comportments*
- ✓ *Concepts communs d'objets pour le développement de l'application et de la plate-forme*
- ✓ *Extension du langage des composants d'UML pour le langage des objets*
- ✓ *Réutilisation des associations UML en restreignant la portée*
- ✓ *Réutilisation des machines à états UML pour la modélisation des comportements*
- ✓ *Extension des transitions UML en termes de réception et d'envoi de messages*
- ✓ *Simplification de la notation graphique des actions des transitions UML*

3 Méta-modèle et phase d'analyse du système

L'objectif de la phase d'*analyse du système* est de décrire le système et le modéliser afin qu'il réponde aux besoins exprimés durant la phase d'*analyse du besoin*. L'activité d'*ouverture hiérarchique* est la même pour le développement d'une application ou d'une plate-forme. Elle consiste à choisir un objet du système vu comme une *boite noire* avec un comportement *apparent*, d'en décrire les objets le constituant et de modéliser son comportement *propre*.

Nous distinguons dans cette phase le comportement *apparent* d'un objet et son comportement *propre*. Le comportement *apparent* est le comportement modélisé lorsque l'objet est vu comme une *boite noire*, c'est-à-dire *fermé*, pouvant contenir d'autres objets qui ne sont pas encore révélés. Le comportement qui lui est associé n'est alors pas son comportement véritable, mais l'agrégation de son comportement *propre* avec les comportements de ses constituants.

L'activité de modélisation durant la phase d'*analyse du système* est *récursive*. Elle s'applique pour l'ouverture initiale du système, ainsi que pour l'ouverture de ses constituants. Le système ouvert présente une structure hiérarchique d'objets le constituant. Tous les objets possèdent un comportement modélisé par une machine à états, à l'exception du système initial dont le comportement *apparent* est décrit par l'agrégation des scénarios formalisés en *analyse du besoin*.

À tout instant, il est possible de vérifier qu'une ouverture d'un objet est conforme au besoin exprimé par un ensemble de scénarios de l'*analyse du besoin*. Cette activité se nomme le *rejoué de scénarios*. Pour rejouer un scénario, le scénario est « lu » et les machines à états des objets sont interprétées. Chaque message envoyé par l'acteur est un événement pouvant déclencher le franchissement d'une transition dans la machine à états du système. Le système peut répondre directement à l'acteur ou bien déléguer les tâches aux objets constituants. Tout le long de l'interprétation des machines à états, une *trace* – modélisée par un diagramme de *séquences UML* – est produite. L'ouverture est considérée comme *valide* si la trace obtenue est *conforme*

au scénario du modèle d'*analyse du besoin*, c'est-à-dire qu'il est possible de substituer dans la trace l'objet ouvert avec l'ensemble de ses constituants par l'objet vu en boite noire et visualiser le même échange ordonné de messages défini dans le scénario.

3.1 Le langage

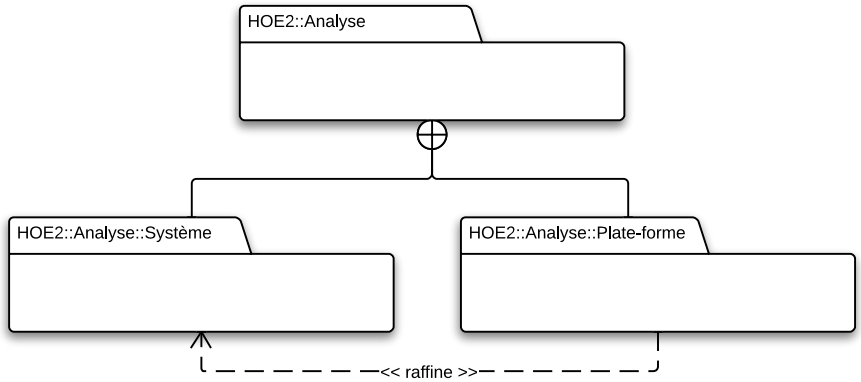


FIGURE 52 – Paquetage de la phase d'analyse du système

Syntaxe abstraite. Bien que les tâches durant la phase d'analyse du système pour le développement d'une application et d'une plate-forme soient les mêmes, les artefacts produits sont différents. Aussi, dans la figure 52 illustrant l'organisation du méta-modèle ⟨HOE⟩² pour la phase d'*analyse du système*, nous avons choisi de séparer la partie du méta-modèle relative à la branche gauche du processus en un paquetage nommé HOE2::Analyse::Système de celle de la plate-forme dans un second paquetage HOE2::Analyse::Plate-forme. Ce dernier paquetage est un raffinement du premier. Ce choix a été fait puisque nous considérons dans ⟨HOE⟩² qu'une plate-forme est une spécialisation d'un système.

La figure 53 illustre le méta-modèle d'*analyse du système*. Ce méta-modèle s'appuie sur le méta-modèle d'*objet* illustré par la figure 46 en page 97. Le concept de *système* est un *objet* particulier qui modélise le *système* dont le besoin a été spécifié durant la phase d'*analyse du besoin*. Le système contient par ailleurs la liste des objets qui le composent. Dans le cas particulier d'une *plate-forme*, les objets sont de différentes natures [Hili *et al.* 2014b]. Les *ressources* regroupent la mémoire, les unités de calcul et les ressources de communication, c'est-à-dire les objets permettant de stocker les données et le code, de transférer les données et d'exécuter le code ; les *périphériques* sont des objets permettant d'interfacer le système avec le monde extérieur. Sur une plate-forme réelle, il peut s'agir de pilotes de périphériques réels (des capteurs et actionneurs), ou des objets de communication (bus, réseau sur puce) ; le *monde* est un concept propre à ⟨HOE⟩², inspiré du PDSI [Rygaert 2002]. Il regroupe les ressources permettant d'héberger et d'exécuter des objets et les alloue aux objets qu'il héberge. Nous pouvons dire qu'un monde est la combinaison d'une mémoire pour stocker les objets et d'une unité de calcul munie de ses règles d'interprétation pour interpréter les messages qui sont contenus dans la machine à états de l'objet. Un monde peut être logiciel ou matériel. Un ordinateur possède un monde matériel qui est la combinaison d'une mémoire physique, généralement le disque dur, pour stocker du code (les objets), d'une mémoire volatile, généralement la mémoire vive, pour

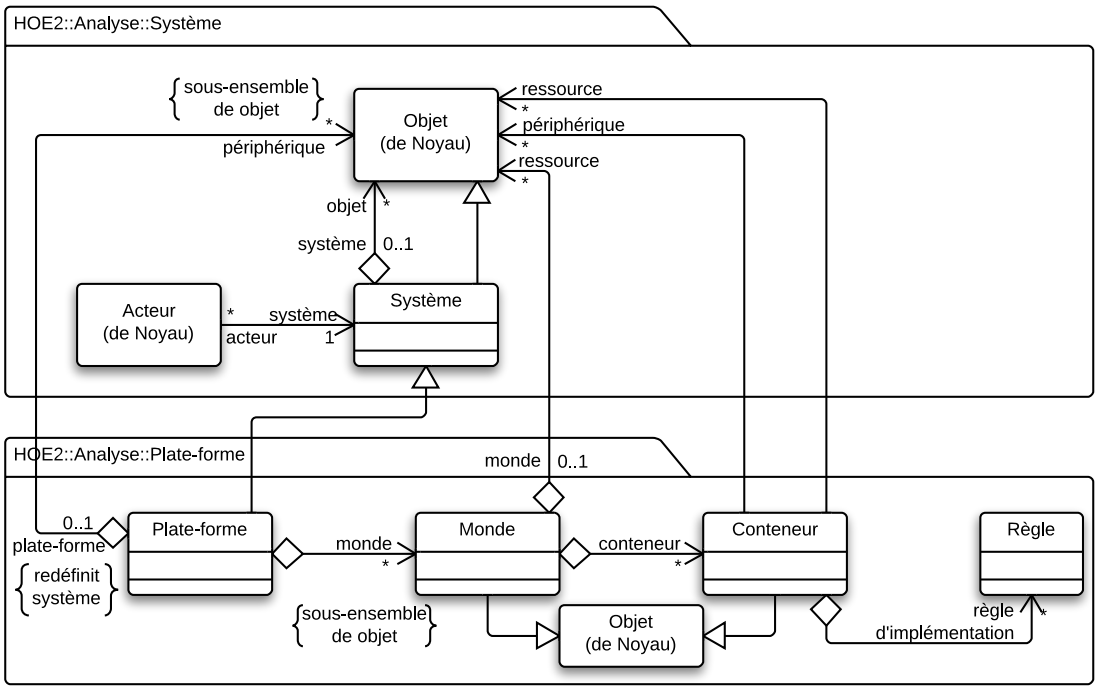


FIGURE 53 – Méta-modèle d'analyse du système

stocker les données et d'un processeur (l'unité de calcul) qui est muni de son jeu d'instructions (les règles d'interprétation) pour exécuter le code. Le dernier concept propre à ⟨HOE⟩[2] est le *conteneur*. Il permet de définir comment sont réellement implémentés les objets dans les mondes et comment les objets implémentés peuvent accéder aux périphériques. Le conteneur joue le rôle d'un élément substituable de la plate-forme, un *réceptacle* de la plate-forme qui est modélisé au sein des mondes et associé aux périphériques de la plate-forme. Il fournit des règles d'implémentation permettant de définir comment des objets seront implémentés durant la phase d'*implémentation du système*.

Exemples et notation. Le tableau 21 liste les quatre types de diagramme utilisés durant la phase d'analyse du système. Les deux premiers diagrammes sont propres à cette phase. Le diagramme de *comportement* permet de manipuler le comportement de chaque objet à tous les niveaux et *a fortiori* n'est pas propre à cette phase. De même, le diagramme de *trace*, permettant de *rejouer* les scénarios du besoin sera réutilisée dans les phases suivantes.

La figure 54 illustre un modèle d'*analyse du système de suivi de passants* pouvant être visualisé dans le diagramme d'*analyse*. Ce système est composé de trois objets, un Détecteur de présence pour détecter une présence dans une zone déterminée, un Détecteur de visage pour détecter un visage et sa position depuis un flux vidéo et un Contrôleur de tourelle pour contrôler l'orientation d'une caméra sur une tourelle pilotable selon deux angles. Ce système est composé d'une ouverture initiale du système et potentiellement de différents compléments d'ouverture, en fonction des scénarios qu'il doit satisfaire. Par exemple, le Contrôleur de tourelle a pu être obtenu dans une première ouverture pour satisfaire le scénario « *Orienter la caméra* » tandis

Diagramme	Description
Diagramme d'analyse	Permet de visualiser le modèle d'analyse du système. Il est le même pour modéliser le besoin et l'application et de la plate-forme. Il présente le système considéré de façon arborescente, le système tout en haut et les objets qui le composent, ces derniers pouvant être eux-même composés.
Diagramme d'ouverture	Permet de réaliser l'activité d'ouverture d'objets. Il est décomposé en deux parties. La première partie permet de visualiser l'objet *fermé*, ainsi que les autres objets nécessaires pour son ouverture. La seconde partie permet de réaliser l'ouverture de l'objet.
Diagramme de comportement	Permet de modéliser le comportement des objets d'analyse. Le *diagramme de comportement*, n'est pas propre à la *phase d'analyse*, mais sera également utilisé durant les autres phases.
Diagramme de trace	Permet de visualiser une trace produite par le *rejoué de scénarios*. Ce diagramme est obtenu automatiquement en rejouant les scénarios du modèle d'*analyse du besoin* dans le contexte d'une ouverture effectuée.

TABLE 21 – Diagramme de la phase d'analyse du système

que le « *Détecteur de visage* » a pu être obtenu dans un complément d'ouverture pour satisfaire des scénarios du cas d'utilisation « *S'abonner à la détection de visage* ».

Les acteurs et les objets sont illustrés selon leurs représentations simplifiées. Les objets sont connectés aux travers d'associations UML. Une association connecte l'acteur au premier objet du modèle qui représente le système. Chaque objet possède un comportement défini par une machine à états. La figure 62 illustre l'usage des diagrammes de *comportement* pour modéliser les comportements des différentes objets du modèle. Cette figure illustre la communication entre les deux objets, modélisée par des envois et des réceptions de messages. Lorsque le système est notifié de la détection d'un visage, il transmet les coordonnées au contrôleur de tourelle. En cas d'absence, il demande à ce dernier de réinitialiser la position de la caméra.

La figure 56 illustre un rejoué de scénario *valide* et la production d'une trace dans le diagramme de *trace*. La trace (fig. 56b) est obtenue en rejouant le scénario nominal « *Orienter*

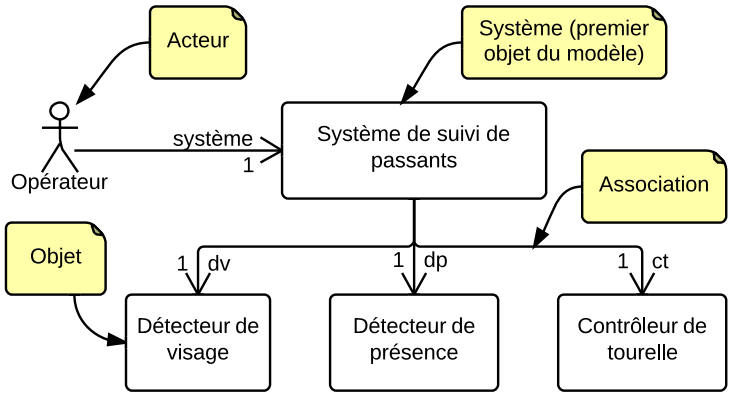
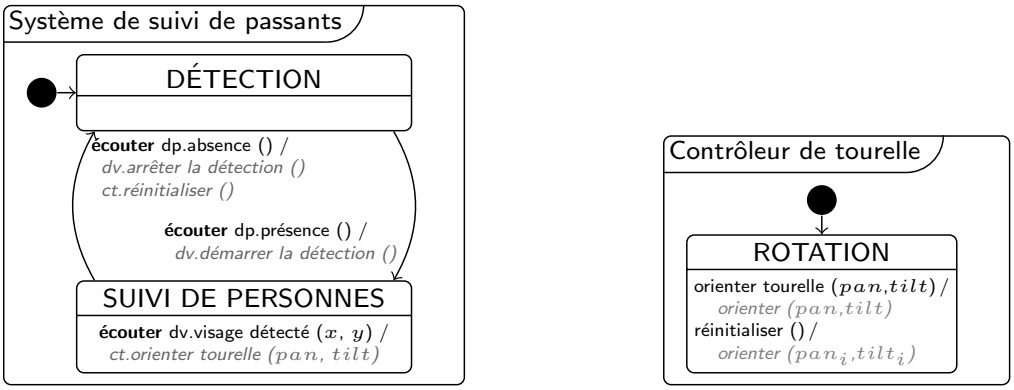
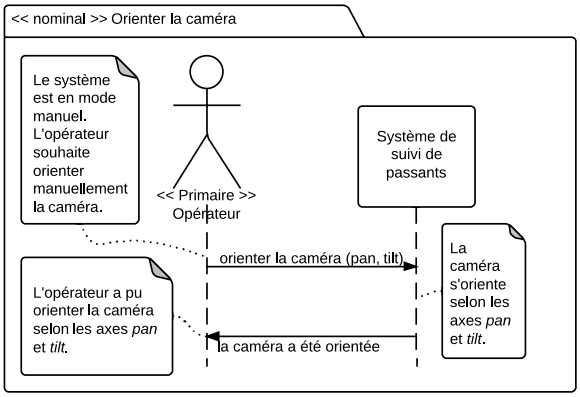


FIGURE 54 – Diagramme d'analyse du système de suivi de passants

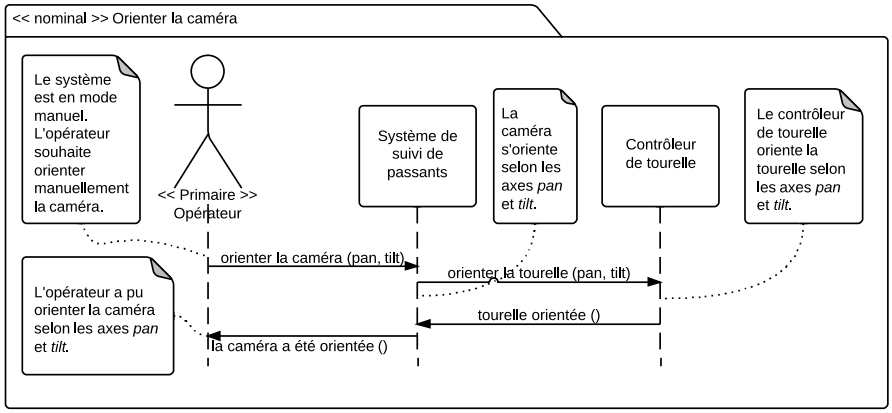


(a) Système de suivi de passants (b) Contrôleur de support

FIGURE 55 – Diagrammes de comportement d'objets du système



(a) Scénario « Orienter la caméra » (niveau analyse du besoin)



(b) Trace obtenue en rejouant le scénario nominal « Orienter la caméra »

FIGURE 56 – Diagramme de trace pour rejouer un scénario d'analyse du besoin

la caméra » (fig. 56a). Le diagramme de *trace* se distingue de diagramme de *scénario* car il fait apparaitre tous les objets constituants qui jouent un rôle dans la communication avec le système. Les messages échangés entre l'acteur Opérateur et le système sont identiques et apparaissent dans le même ordre, qui prouve la conformité de la trace par rapport au scénario et valide donc l'ouverture par rapport au besoin.

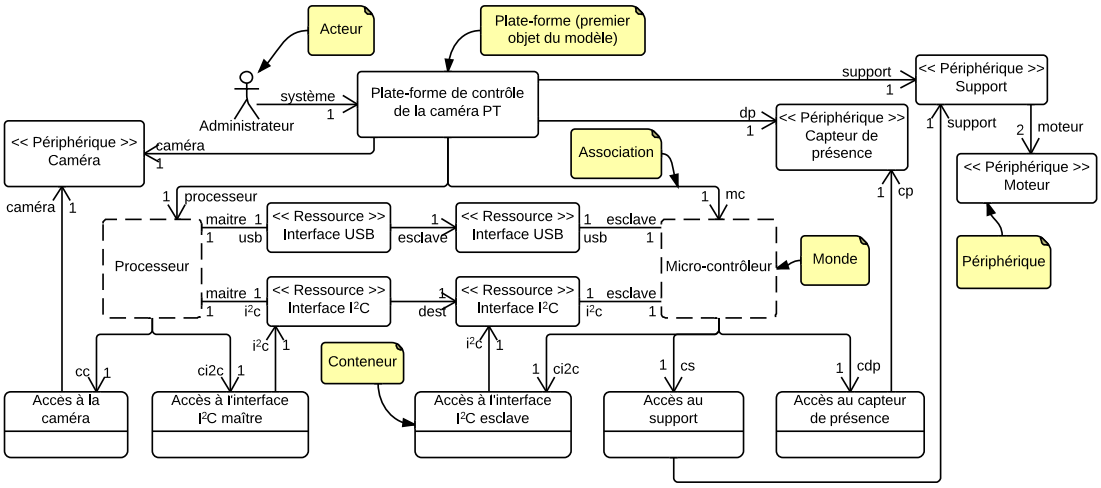
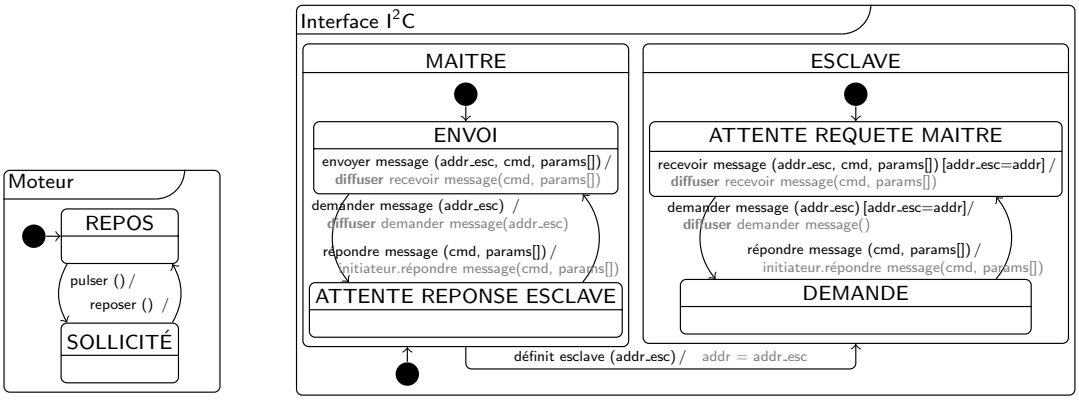


FIGURE 57 – Diagrammes d'analyse de la *plate-forme de contrôle de la caméra PT*

Pour modéliser la plate-forme, nous utilisons le même diagramme que pour modéliser l'application avec une notation particulière pour les différents éléments de la plate-forme. La figure 57 illustre le modèle d'*analyse de la plate-forme de contrôle de la caméra PT*. Cette plate-forme permet à une application qu'elle héberge de piloter une *caméra*, montée sur un *support* qui s'oriente suivant deux axes à l'aide de deux *moteurs*. Le modèle est composé de deux mondes, Processeur et Micro-contrôleur, illustrés par des rectangles à trait discontinu, pour symboliser leur capacité à délimiter une zone dans laquelle des objets sont hébergés. Ils sont munis de leurs ressources (non représentées) en termes de calcul et de mémoire ainsi que des ressources de communication (des interfaces au bus série USB [5] et au bus I^2C [6], identifiés sur la figure par les stéréotypes « Ressource »). Ces deux mondes fournissent des conteneurs, permettant les accès aux ressources de communication et aux différents périphériques de la plate-forme. Les conteneurs sont illustrés par des rectangles à deux compartiments. Le premier compartiment contient le nom du périphérique tandis que le second illustre l'emplacement dans lequel un objet sera injecté durant la phase d'*implémentation du système*.

Les périphériques de la plate-forme sont, à l'instar des ressources, identifiés par le stéréotype « Périphérique ». Nous avons modélisé trois périphériques, une caméra, un support et un capteur de présence. Le support est également détaillé et constitué de deux moteurs. Les trois conteneurs permettent l'accès à ces différents périphériques pour les objets qu'ils hébergent.

5. En anglais, Universal Serial Bus (USB).
6. En anglais, Inter Integrated Circuit (I^2C).



(a) Moteur (b) Interface I²C

FIGURE 58 – Diagrammes des comportements apparents des ressources et périphériques

La figure 58 illustre l'usage du diagramme de *comportement* pour modéliser les comportements du moteur (à gauche) et de l'interface I²C (à droite). Le moteur est un *servomoteur* asservi en position. Une pulsation équivaut à une sollicitation endurée par le moteur qui tourne à vitesse fixe. Le moteur alterne entre un état REPOS et un état SOLLICITÉ, la durée d'alternance étant définie par une durée de pulsation. La durée de pulsation permet de contrôler l'orientation du moteur. L'interface I²C possède deux états, MAITRE et ESCLAVE. En mode MAITRE, l'interface est en mesure d'envoyer un message à une autre interface configurée en mode ESCLAVE et identifiée par une adresse *addr_esc*. Initialement, une interface *esclave* est en attente d'une requête de l'interface maître. Elle ne peut que répondre à des messages de ce dernier et ne peut pas initier une conversation. La présence du mot-clé *initiateur* montre l'usage du concept de *réponse* du paquetage Noyau::Comportement. Les composants de la plate-forme étant des spécialisations des objets d'un système, nous avons fait le choix de conserver la même notation pour la modélisation des comportements des objets.

Id	Description
[1]	Les acteurs du système ne peuvent communiquer qu'avec le système et non un objet composant le système .

TABLE 22 – Règle de cohérence intra-modèle pour l'analyse d'une application

Règles de cohérence intra-modèle. Cette partie fournit les règles de cohérence pour la construction des modèles durant la phase d'analyse, qui viennent s'ajouter aux règles de cohérence du méta-modèle *noyau*. Les contraintes propres à cette phase sont répertoriées dans les tableaux 22 et 23. Pour le système en général, la première règle s'assure de la construction hiérarchique du système avec les acteurs dialoguant avec le système. Cela permet notamment d'assurer que le rejoué de scénario s'effectue bien. Pour la plate-forme, les règles permettent de guider sa construction. La seconde et la troisième règle définissent le rôle joué par un objet vis-à-vis de la plate-forme selon qu'il est directement contenu dans l'un ou l'autre. La dernière règle contraint les associations du conteneur avec les ressources et périphériques de la plate-

Id	Description
[2]	Un objet de la plate-forme joue le rôle de *périphérique* lorsqu'il est directement contenu par la plate-forme.
[3]	Un objet de la plate-forme joue le rôle de *ressource* lorsqu'il est directement contenu par un monde de la plate-forme.
[4]	Un conteneur ne peut être associé qu'aux périphériques de la plate-forme et aux ressources du monde auquel il appartient.

TABLE 23 – Liste des règles de cohérence intra-modèle pour l'analyse d'une plate-forme

forme. Notamment, un conteneur fourni dans un monde ne peut accéder aux ressources d'un autre.

Règles de cohérence inter-modèles. Les règles présentées dans cette partie permettent d'assurer la cohérence entre le modèle développé lors de la phase d'analyse du besoin et celui de l'analyse du système. Le tableau 24 liste ces règles. Les deux premières s'assurent que les acteurs et le *système considéré* du modèle d'analyse du besoin possèdent leurs homologues dans le modèle d'analyse du système. Les deux dernières s'assurent que tous les messages intervenant dans les scénarios du modèle d'analyse du besoin alimentent les interfaces des acteurs et du premier objet du système dans le modèle d'analyse du système.

Id	Description
[1]	Tous les *acteurs* de modèle d'analyse du besoin correspondent à des *acteurs* dans le modèle d'analyse du système.
[2]	le *système considéré* du modèle du besoin correspond au premier *objet* du modèle d'analyse du système.
[3]	Tous les messages reçus et émis par le *système considéré* au travers des *scénarios* du modèle d'analyse du besoin sont contenus dans l'ensemble des interfaces requises et fournies par le premier *objet* du modèle d'analyse du système.
[4]	Tous les messages reçus et émis par un *acteur* au travers des *scénarios* du modèle d'analuse du besoin sont contenus dans l'ensemble des interfaces requises et fournies par l'*acteur* présent dans le modèle d'analyse du système.

TABLE 24 – Liste des règles de cohérence inter-modèles de l'analyse du système

3.2 La démarche

La figure 59 illustre les différentes tâches durant la phase d'*analyse du système*. Le point d'entrée de la phase est un modèle d'*analyse du besoin* CONSISTANT ou COMPLET. Le fait d'accepter comme point d'entrée un modèle du besoin CONSISTANT permet par exemple de fixer quelques besoins prioritaires et de modéliser le système en phase d'*analyse du système* tout en continuant l'*analyse du besoin*. Par exemple, cette spécificité du processus ⟨HOE⟩² permet de traiter la réalisation de prototypes.

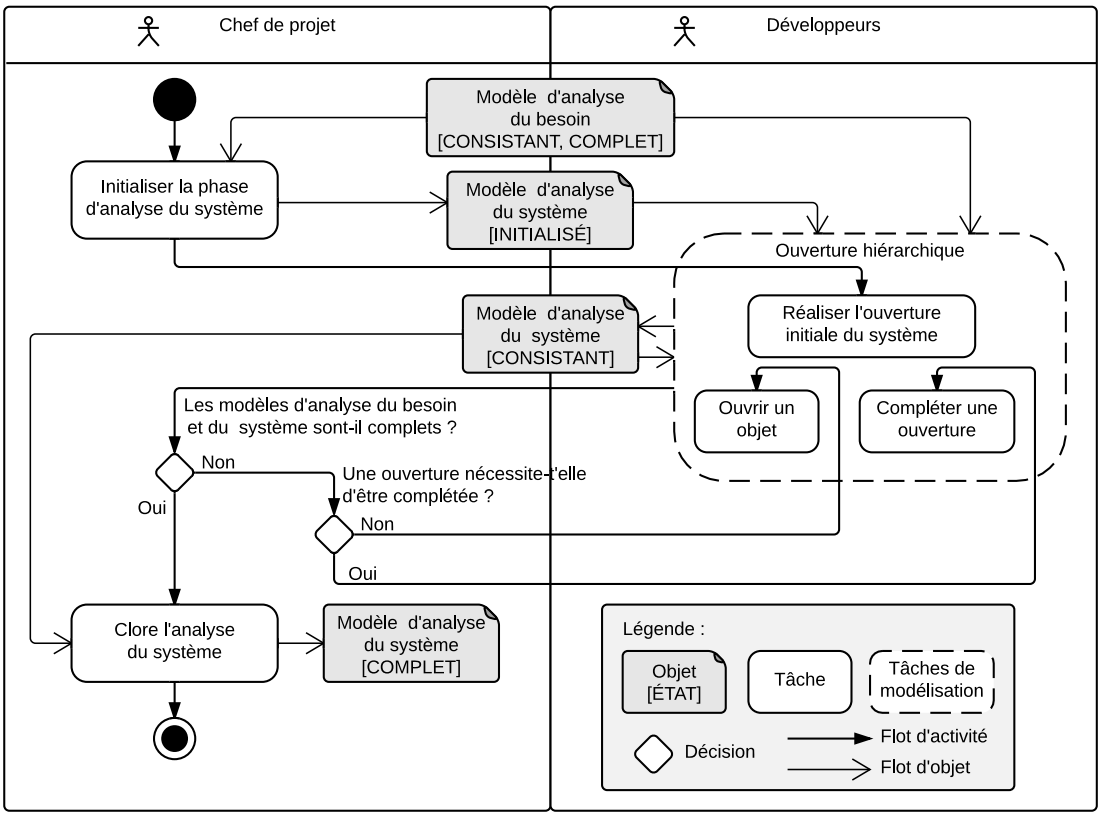


FIGURE 59 – Diagramme d'activités modélisant l'activité d'ouverture hiérarchique

Initialisation de la phase. Lors de l'initialisation de la phase d'analyse du système, un *modèle d'analyse du système* est créé, dans l'état INITIALISÉ. Il contient le premier *objet* représentant le *système* et l'ensemble des *acteurs*. La figure 60 illustre le modèle d'*analyse du système de suivi de passants* dès son *initialisation*. Dans ce modèle, le système est initialisé avec l'ensemble des messages qu'il est capable d'émettre et recevoir. La liste des acteurs est également remonté du modèle d'*analyse du besoin*. Pour chacun, une association en provenance de l'acteur vers le système est créée. Ce système ne contient pas encore de machine à états modélisant son comportement, ce dernier étant défini par l'ensemble des scénarios formalisés en *analyse du besoin*.

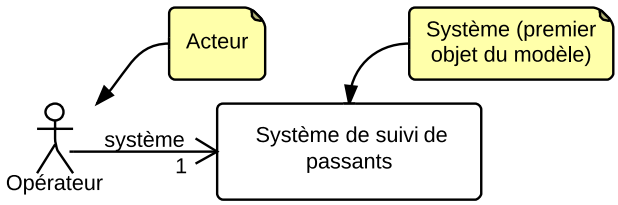
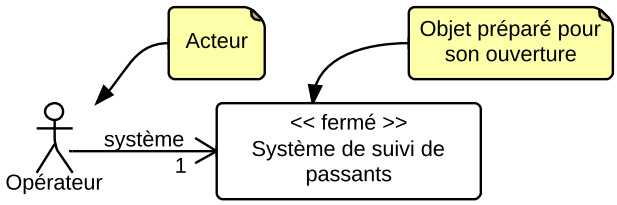
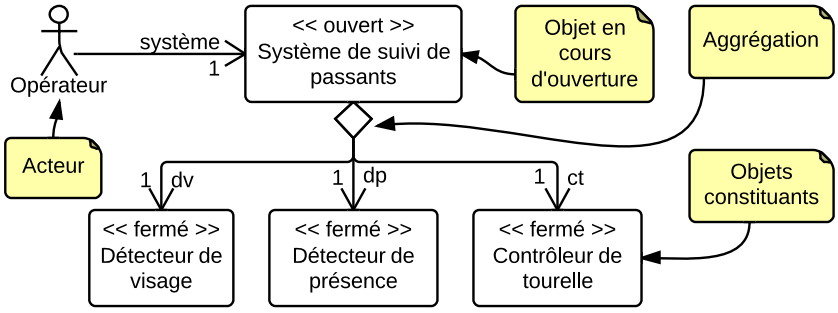


FIGURE 60 – Système initialisé dans le modèle d'analyse du système



(a) Objet fermé avant ouverture (non éditable)



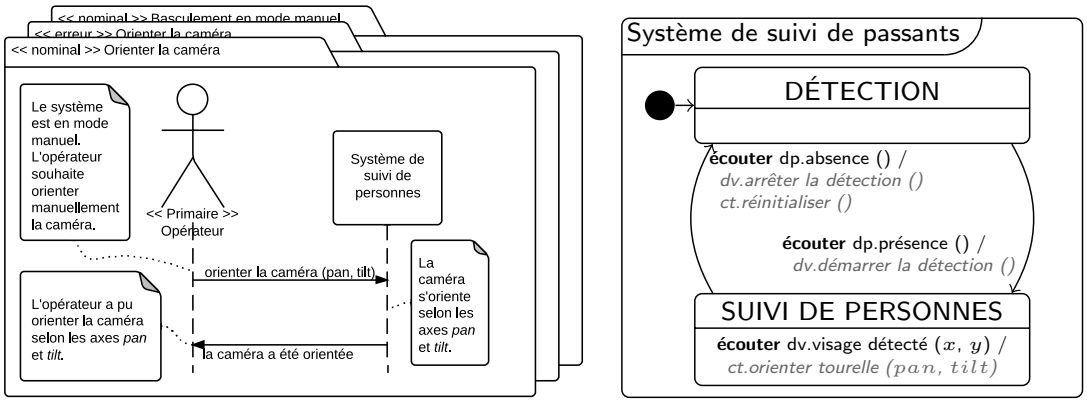
(b) Objet ouvert après ouverture (éditable)

FIGURE 61 – Diagramme d'ouverture avec l'ouverture initiale du système

Tâches de modélisation « *Ouverture hiérarchique* ». L'*ouverture hiérarchique* consiste à ouvrir et à détailler le contenu d'un objet *fermé*, en faisant apparaitre de nouveaux objets le constituant et de nouvelles associations. Cette ouverture se décline en trois tâches : *ouverture initiale du système*, *ouverture d'un objet* et *complément d'ouverture*. La figure 61 illustre une ouverture (ici l'ouverture initiale du système) dans un diagramme d'*ouverture*. La partie haute de la figure reprend le système et ses acteurs. Elle est utilisée à des fins de visualisation. La partie basse permet de réaliser l'ouverture, en remplaçant le système fermé avec un comportement *apparent* par un système ouvert avec un comportement *propre* et des objets constituants. Une spécificité réside dans l'utilisation de la *notation graphique* de relation d'*agrégation* UML pour décrire l'attachement à l'objet ouvert des objets qui le composent ainsi que des deux stéréotypes « ouvert » et « fermé ». Ces trois éléments sont purement visuels et ne constituent pas des éléments issus du méta-modèle d'*analyse du système*. Ils décrivent l'état des objets dans le contexte d'une ouverture hiérarchique. Nous reviendrons sur cet aspect en abordant la gestion de projet et les *feuilles de tâche* dans le chapitre 8.

L'*ouverture initiale du système* se distingue de l'*ouverture hiérarchique d'un objet* au sens où le système est le premier objet du modèle et son comportement *apparent* n'est défini que par l'ensemble des scénarios du *modèle d'analyse du besoin* mis bout à bout (cf. fig. 62a). Lors de cette première ouverture, le développeur remplace alors le système par un objet ouvert avec un comportement *propre* (cf. fig. 62b) et ses constituants. Les comportements des constituants sont modélisés en termes de machines à états, décrites par les figures 55b et 63.

Cette ouverture se termine lorsque le premier niveau d'ouverture a été effectué. Dans le cas de l'*ouverture d'un objet*, l'objet *fermé* possède cette fois-ci un comportement *apparent* modélisé à l'aide d'une machine à états. Le travail du développeur consiste alors à substituer le compor-

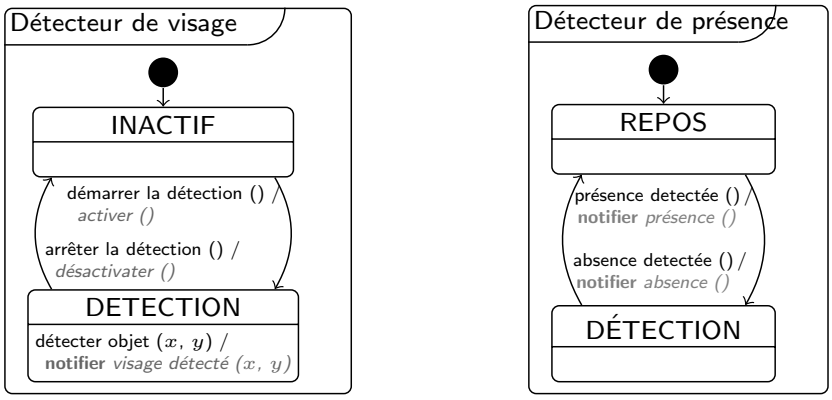


(a) Comportement apparent (avant ouverture)

(b) Comportement propre (après ouverture)

FIGURE 62 – Comportement du système avant et après ouverture

tement *apparent* de l'objet par son comportement *propre* et l'ensemble des comportements des objets le constituant.



(a) Détecteur de visage

(b) Détecteur de présence

FIGURE 63 – Comportement des objets constituants du système

Dans les deux cas, l'ouverture est conditionnée à la couverture d'une largeur du besoin, c'est-à-dire que l'ouverture d'un objet pour modéliser son comportement et des constituants doit satisfaire un ensemble de scénarios modélisés dans la phase d'analyse du besoin. Une troisième tâche, le *complément d'ouverture*, permet de reprendre une ouverture déjà réalisée pour satisfaire plus de scénarios. Durant cette activité, le développeur ne *substitue* pas le comportement *apparent* d'un objet par son comportement *propre* mais *modifie* le comportement *propre* de l'objet déjà obtenu afin de satisfaire de nouveaux scénarios *tout en respectant les scénarios déjà satisfaits*. Cette spécificité distingue cette tâche de celle d'*ouverture*. Le développeur peut modéliser de nouveaux objets afin de satisfaire les nouveaux scénarios.

Clôture de la phase. La phase peut être close dès le moment où le modèle d'analyse du besoin est lui-même clos. Dans ce cas précis, et dans le cas où tous les scénarios de l'analyse du besoin sont satisfaits par la modélisation de l'analyse du système, alors cette phase peut également être close. La clôture de la phase a pour effet de passer le modèle d'analyse du système dans l'état COMPLET, et il n'est plus possible de le modifier.

En résumé

- ✓ *Tâche d'ouverture hiérarchique pour décrire le contenu du système vu en boite noire*
- ✓ *Mêmes tâches de modélisation pour l'analyse d'une application ou d'une plate-forme*
- ✓ *Méta-modèle d'analyse du système général permettant la modélisation d'un système*
- ✓ *Méta-modèle d'analyse de la plate-forme raffinant le méta-modèle d'analyse du système*
- ✓ *Introduction des concepts de monde, conteneur, ressource et périphérique de la plate-forme*
- ✓ *Initialisation de la phase dès que le modèle de l'analyse du besoin est* CONSISTANT
- ✓ *Rejouer du scénario pour valider l'ouverture hiérarchique et assurer la traçabilité au besoin*

4 Méta-modèle et phase de conception du système

La phase de *conception du système*[7] permet d'introduire une première définition de la plate-forme dans le flot de développement de l'application. Cette première définition est limitée à l'introduction des *mondes* de la plate-forme. Durant cette phase, le développeur de l'application effectue les premiers choix d'implémentation. Ces choix se limitent à définir où (dans quels mondes) seront hébergés les objets de l'application obtenus durant la *phase d'analyse*. Durant cette phase, les objets modélisés en *analyse du système* sont alors « *découpés* » en *fragments d'objets*, chaque fragment est ensuite *hébergé* sur un monde de la plate-forme. Nous ferons par la suite la distinction entre les *objets d'analyse*, modélisés durant la phase d'*analyse du système* et les *objets de conception*, représentant des fragments des objets d'analyse.

Bien que les activités de découpage des objets durant la phase de *distribution* et d'ouverture dans la phase précédente se ressemblent, elles sont en réalité radicalement différentes. La distribution est une activité spécifique à la conception des systèmes embarqués où l'une des problématiques fondamentales réside dans la fragmentation du code en vue de son exécution dans différentes parties de la plate-forme. Ce choix peut être dû à un ensemble de raisons : optimisation et rapidité d'exécution, réduction de la consommation, minimisation de la plate-forme physique, fiabilité, redondance, parallélisme, etc. À ce titre, la distribution effectuée durant la phase de *conception du système* relève de choix d'implémentation tandis que l'ouverture hiérarchique effectuée durant la phase précédente relève d'un découpage fonctionnel des objets, ce qui distingue foncièrement ces deux activités.

7. Jusqu'à présent, nous parlions de modèle d'*analyse du système*. Durant les phases de conception et d'implémentation, le processus se poursuit uniquement sur la branche applicative. Pour cela, nous parlerons désormais de modèles d'analyse de l'*application* et de la *plate-forme*. Les modèles suivants traduisent l'implémentation des objets *applicatifs* sur la *plate-forme*, l'ensemble constituant le système embarqué. Pour cette raison, nous appellerons les deux prochains modèles les modèles de conception et d'implémentation *du système*

4.1 Le langage

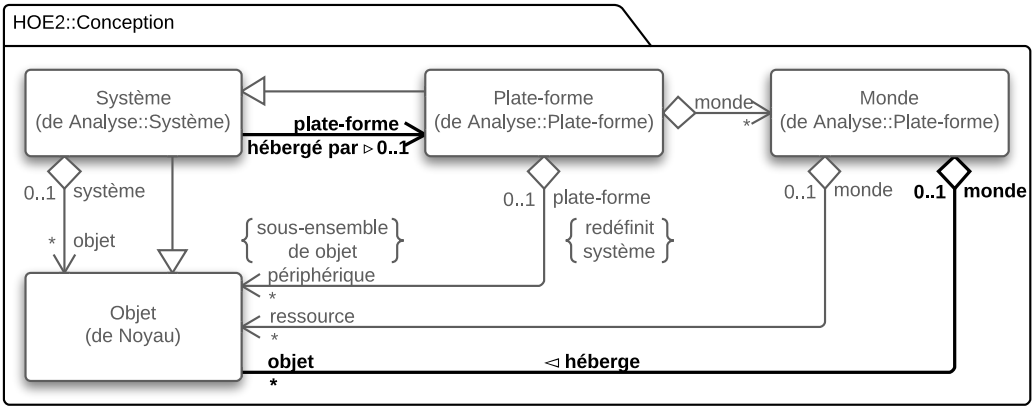


FIGURE 64 – Méta-modèle de conception

Syntaxe abstraite. La figure 64 illustre le méta-modèle de *conception du système*. Il est proposé comme un *raffinement* du méta-modèle d'*analyse du système*. Il permet de faire le lien entre l'application et la plate-forme, toutes deux modélisées durant la phase d'analyse du système. Ce méta-modèle ajoute deux nouvelles associations, modélisant l'*hébergement* de l'application sur la plate-forme et l'*hébergement* des objets de conception dans les mondes de la plate-forme.

Diagramme	Description
Diagramme de conception	Permet d'afficher les différents mondes de la plate-forme et d'y glisser les objets applicatifs, en les découpant éventuellement préalablement.
Diagramme de distribution	Similaire au diagramme d'ouverture durant la phase d'analyse du système (cf. table 21 de la page 106). Permet de réaliser une *distribution*.
Diagramme de comportement	Voir table 21 de la page 106.
Diagramme de trace	Voir table 21 de la page 106.

TABLE 25 – Diagrammes de la phase de conception du système

Exemples et notation. Le tableau 25 liste les quatre diagrammes utilisés durant la phase de conception du système. En plus des diagrammes de *comportement* et de *trace* déjà employés dans la phase précédente, deux nouveaux diagrammes, les diagrammes de *conception* et de *distribution* sont utilisés durant la phase de *conception du système*.

La figure 65 illustre la distribution du système qui a été ouvert dans la figure 61b. Le choix a été fait d'héberger les objets Détecteur de visage, Détecteur de présence et Contrôleur de tourelle directement dans les mondes Processeur et Micro-contrôleur sans les découper. En revanche, l'objet du système a été découpé en deux et réparti sur les deux mondes. Pour ce découpage, certaines précautions ont dû être prises. Dans le modèle d'*analyse de l'application*

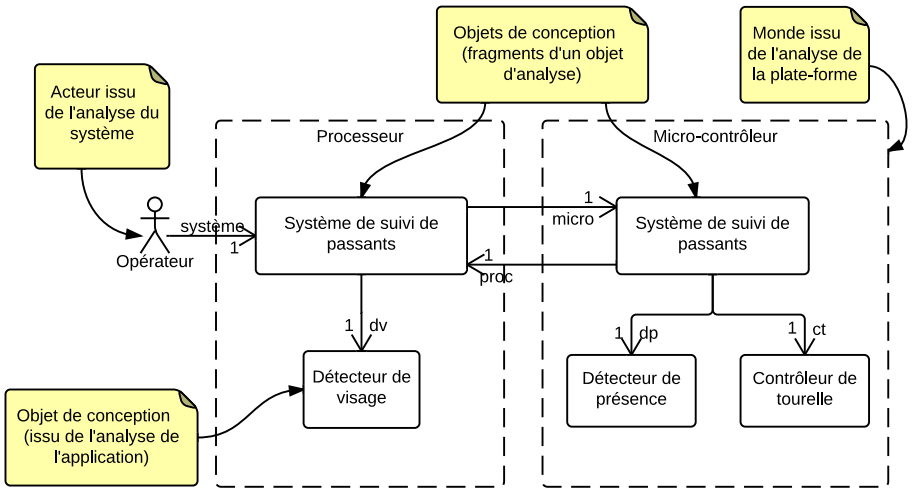
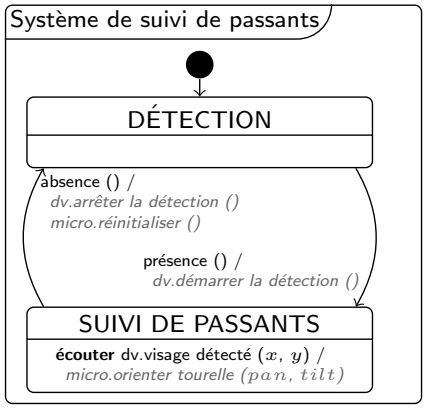
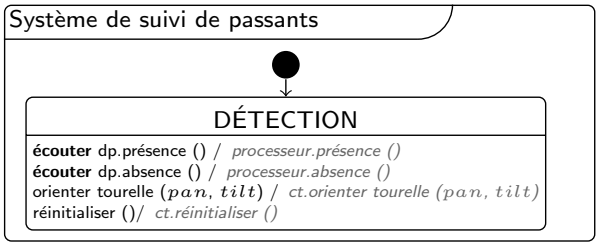


FIGURE 65 – Diagramme de distribution du système de suivi de passants

obtenu après l'ouverture hiérarchique de la figure 61b, il n'existait aucune association entre le Détecteur de visage et le Détecteur de présence. Cette association ne peut alors être modélisée en phase de *conception du système*. Par ailleurs, le système a été découpé en deux en conception. La modélisation d'au moins une association entre ces deux objets est donc obligatoire. Enfin, le système pouvait communiquer avec tous les autres objets. Créer de nouvelles associations dans le modèle de conception, par exemple entre la partie du système hébergée par le Micro-contrôleur et le Détecteur de visage serait donc autorisé et correspondrait à des choix d'ingénierie différents de ceux que nous avons effectués pour modéliser le système.



(a) Fragment hébergé par le Processeur



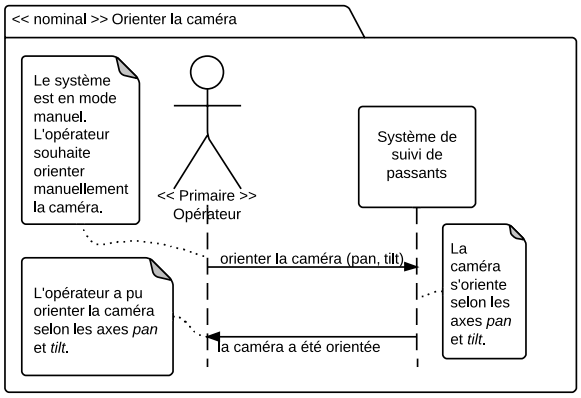
(b) Fragment hébergé par le Micro-contrôleur

FIGURE 66 – Comportements des fragments du système répartis sur les deux mondes

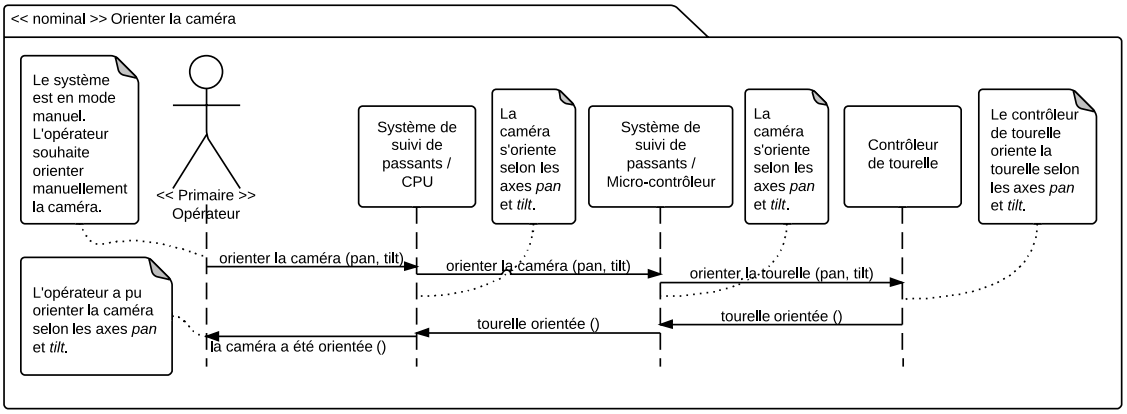
La figure 66 illustre les comportements des fragments du système. Les autres objets n'ayant pas été découpés, leurs comportements restent inchangés. Dans la figure 66, l'état SUIVI DE PASSANTS a été intégralement « déposé » dans la partie hébergée dans le Processeur tandis

que l'état DÉTECTION a été « découpé » dans les deux fragments : dans cet état, le fragment hébergé sur le Processeur n'effectue aucune activité et est en attente du message *présence* pour entamer le suivi de passants. Il quitte l'état SUIVI DE PASSANTS lorsque le fragment du système hébergé sur le Micro-contrôleur détecte et prévient d'une absence.

On constate la validité de la distribution par la conformité des traces par rapport aux scénarios définis en *analyse du besoin*, c'est-à-dire l'équivalence des échanges de messages entre l'acteur et le système. La figure 67 illustre la trace produite dans le cas de la phase de conception pour le scénario nominal « *Orienter la caméra* ».



(a) Scénario « Orienter la caméra » (niveau analyse du besoin)



(b) Trace obtenue en rejouant le scénario nominal « Orienter la caméra »

FIGURE 67 – Diagramme de trace pour rejouer un scénario en phase de conception

Règles de cohérence intra-modèle. Le méta-modèle de *conception du système* étant un raffinement du méta-modèle d'*analyse du système*, les règles de cohérence de ce dernier s'appliquent en phase de *conception du système*. Le tableau 26 liste ces règles. La première règle permet d'assurer que tous les objets d'analyse sont hébergés dans les mondes de la plate-forme. Les trois dernières règles régissent le découpage et de distribution des objets sur les mondes de la plate-forme.

Id	Description
[1]	Les objets contenus dans le modèle de conception sont obligatoirement hébergés dans les mondes de la plate-forme référencée par le système en conception.
[2]	Si un objet est découpé en plusieurs fragments durant l'activité de distribution, chaque fragment doit porter le même nom que l'objet initial.
[3]	Les fragments d'un même objet doivent être répartis dans des mondes différents.
[4]	Deux ou plusieurs fragments d'un objet hébergés dans des mondes différents doivent pouvoir communiquer entre eux aux travers d'associations.

TABLE 26 – Liste des règles de cohérence intra-modèle pour la conception du système

Id	Description
[1]	À un objet d'analyse correspond un ou plusieurs objets de conception (fragments) dans le modèle de conception du système. Les fragments d'objet portent le même nom que l'objet d'analyse.
[2]	À une association entre deux objets dans le modèle d'analyse de l'application correspond au moins une association entre des fragments des mêmes objets dans le modèle de conception du système.
[3]	Un monde du modèle d'analyse de la plate-forme correspond à un monde du *modèle de conception*

TABLE 27 – Liste des règles de cohérence inter-modèles pour la conception du système

Règles de cohérence inter-modèles. Le tableau 27 résume les règles de cohérence entre les modèles d'*analyse de l'application*, d'*analyse de la plate-forme* et le modèle de *conception du système*. Elles permettent d'assurer que tous les éléments du modèle d'*analyse de l'application* et les mondes du modèle d'*analyse de la plate-forme* sont bien présents dans le modèle de *conception du système*. La première règle permet de s'assurer qu'un objet d'analyse découpé en conception donne lieu à plusieurs fragments portant le même nom, permettant de garder la traçabilité des objets par rapport à la phase d'*analyse du système*. La seconde assure la même traçabilité au niveau des associations. La dernière s'assure que tous les mondes du modèle d'*analyse de la plate-forme* sont bien importés dans le modèle de *conception du système*.

4.2 La démarche

La figure 68 illustre les différentes étapes de la phase de conception. Afin d'entamer cette phase, les points d'entrée sont les modèles d'*analyse de l'application* et *de la plate-forme* sur laquelle seront hébergés les objets de l'application. Ces modèles doivent tous deux être dans l'état CONSISTANT ou TERMINÉ pour pouvoir initialiser la phase de *conception du système*.

Initialisation de la phase. La première tâche consiste à *initialiser* un modèle de *conception du système* en important les *mondes* de la plate-forme qui sont obtenus durant la phase d'*analyse du système*. Ces mondes sont récupérés dans le modèle d'*analyse de la plate-forme*. Le modèle de *conception du système* est alors INITIALISÉ. La figure 69 illustre le modèle de *conception du système* dès son *initialisation*. Dans ce modèle, les mondes sont importés depuis le modèle

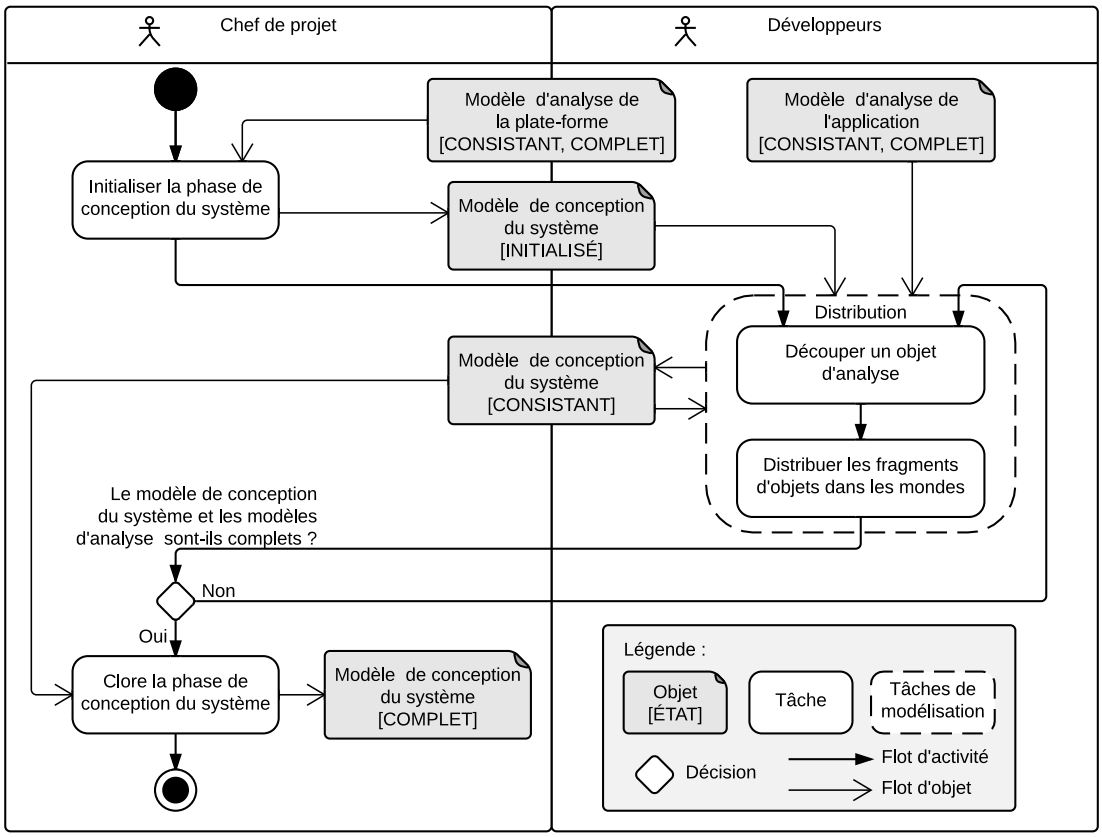


FIGURE 68 – Diagramme d'activités modélisant l'activité de distribution

d'*analyse de la plate-forme* et les acteurs depuis le modèle d'*analyse de l'application*.

Tâches de modélisation « *Distribution* ». La tâche de *distribution* consiste à choisir un objet obtenu dans le modèle d'*analyse de l'application* et de le placer dans un monde. Préalablement, le développeur peut faire le choix de *découper* l'objet et de placer les *fragments*

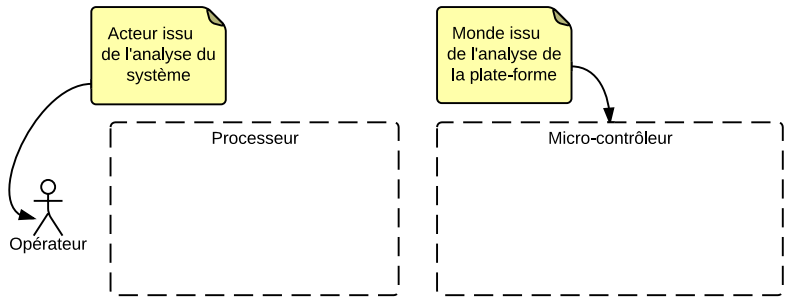
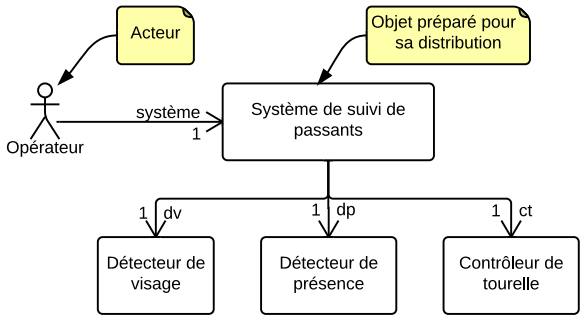
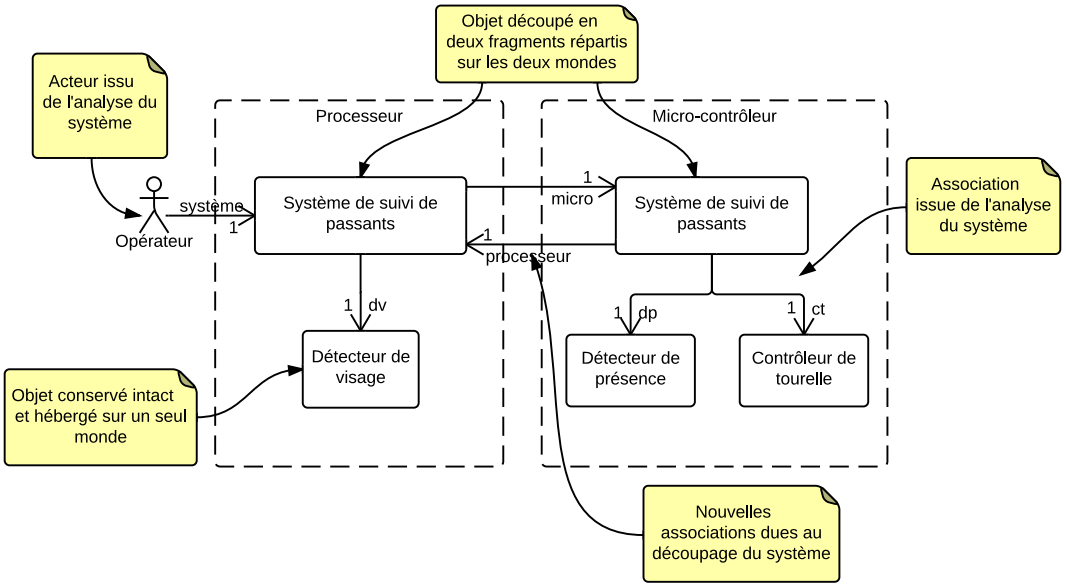


FIGURE 69 – Système initialisé dans le modèle de conception du système



(a) Objet avant distribution (non éditable)



(b) Objet distribué (éditable)

FIGURE 70 – Diagramme de distribution du système de suivi de passants

d'objets dans des mondes différents. Le découpage d'un objet permet d'obtenir des fragments qui seront alors distribués dans des mondes différents. Cela implique de substituer le comportement de l'objet d'analyse par les comportements des morceaux d'objets. La validation du découpage s'effectuant en rejouant les scénarios du besoin pour vérifier qu'ils sont toujours satisfaits.

La figure 70 illustre le diagramme de *distribution* permettant de réaliser la tâche de distribution des objets. Tout comme le diagramme d'*ouverture*, celui de *distribution* est en deux parties, la partie haute non éditable présentant l'objet à distribuer sur les différents mondes de la plate-forme et la partie basse permettant de réaliser la distribution. Une fois la distribution réalisée, il est nécessaire de modéliser les comportements des fragments d'objets. Ces comportements sont illustrés dans la figure 66. Les autres objets n'ont pas été découpés, les comportements modélisés lors de l'analyse de l'application ont donc été conservés. Une fois

une première distribution effectuée, le modèle de conception du système devient CONSISTANT, c'est-à-dire qu'il contient au moins un objet de conception (issu d'un objet d'analyse) hébergé dans un ou plusieurs mondes de la plate-forme. Une nouvelle itération peut commencer, jusqu'à ce que tous les objets d'analyse soient hébergés dans les mondes de la plate-forme.

Clôture de la phase. La clôture de la phase n'est possible que lorsque tous les modèles des phases amont sont dans l'état COMPLET et tous les scénarios doivent pouvoir être satisfaits. Dans ce cas-là, il est possible de clore cette phase ce qui a pour effet de faire passer le modèle de conception du système dans l'état COMPLET.

EN RÉSUMÉ

- ✓ *Activité adressant la problématique d'hébergement d'applications sur les plates-formes*
- ✓ *Intégration partielle du modèle d'analyse de la plate-forme limité aux mondes*
- ✓ *Activité de découpage et de distribution des objets sur les mondes de la plate-forme*
- ✓ *Initialisation de la phase dès que le modèle de l'analyse de l'application est* CONSISTENT
- ✓ *Rejoué du scénario permettant de valider la distribution*

5 Méta-modèle et phase d'implémentation du système

La phase d'*implémentation du système* permet de fournir une description plus détaillée de la plate-forme, permettant de définir comment seront implémentés les objets en accédant aux ressources et périphériques de la plate-forme. Rappelons que l'ensemble des éléments (mondes, ressources, périphériques et conteneurs) de la plate-forme sont modélisés durant la phase d'*analyse de la plate-forme*, mais que seuls les mondes sont importés dans le modèle de *conception du système* (cf. fig. 38 à la page 38). Le modèle d'*implémentation du système* raffine le modèle de *conception du système* en important l'*intégralité* du modèle d'*analyse de la plate-forme*.

L'activité d'implémentation qui a lieu durant cette phase permet au développeur d'effectuer de nouveaux choix d'ingénierie, succédant à ceux (découpage et distribution) qu'il a réalisés en phase de *conception du système*. Il doit durant cette phase indiquer comment il souhaite implémenter concrètement les objets sur les mondes de la plate-forme, en choisissant dans quels conteneurs il souhaite les implémenter. L'implémentation dans les bons conteneurs permettent d'exprimer comment les objets sont concrètement implémentés dans la plate-forme et à ce titre peuvent utiliser les ressources et accéder aux périphériques de celle-ci. Cette tâche est composée de deux parties. La première est la *sélection* d'un ou plusieurs conteneurs dans lequel l'objet est injecté. Elle est *manuelle* et traduit les choix d'implémentation du développeur. La seconde est *automatique* et consiste à *exécuter* les règles d'implémentation des conteneurs afin de transformer l'objet en une implémentation adaptée pour la plate-forme.

5.1 Le langage

Syntaxe abstraite. La figure 71 illustre le méta-modèle d'*implémentation du système*. Il est proposé comme un raffinement du méta-modèle de *conception du système*. Il permet d'intégrer

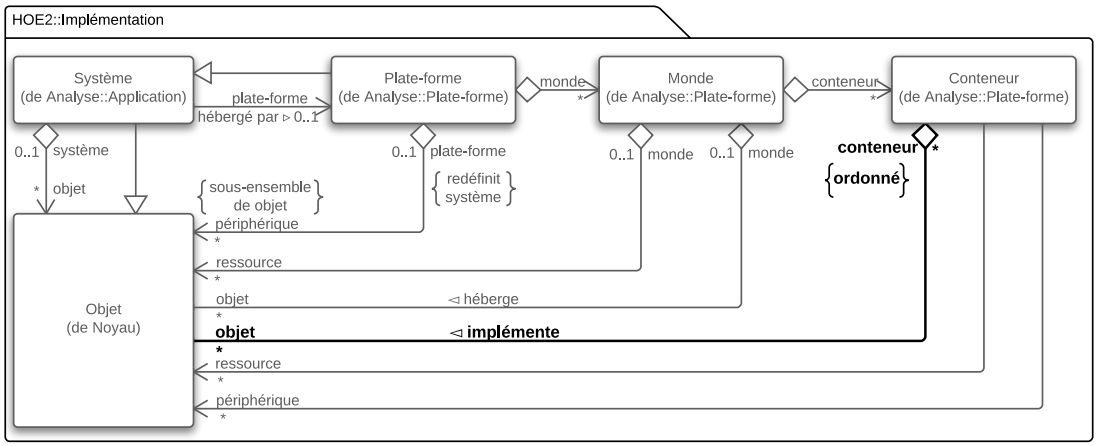


FIGURE 71 – Méta-modèle d'implémentation

l'ensemble du modèle d'*analyse de la plate-forme* dans le flot de développement de l'application. L'ajout dans ce méta-modèle est l'association d'*implémentation* liant les objets modélisés en *conception du système* aux conteneurs de la plate-forme dans lesquels ils sont hébergés. L'injection dans les conteneurs est ordonnée. Dans le cas de l'injection dans plusieurs conteneurs d'un même objet, les règles d'implémentation s'exécutent dans l'ordre d'injection.

Nous avons fait le choix d'autoriser l'injection dans plusieurs conteneurs. Concrètement, ce choix peut mener à des conflits (injection dans deux conteneurs fournissant des règles similaires) pouvant remettre en cause les modélisations des comportements des modèles. Néanmoins, la modélisation des objets devraient être en accord avec le *principe de responsabilité unique*[8] [Martin 2002] où chaque objet devrait n'avoir qu'une seule responsabilité et donc n'utiliser qu'un unique ensemble de ressources et périphériques accessibles par un unique conteneur. Dans le *système de suivi de passants*, un objet injecté à la fois dans les conteneurs d'accès à l'interface I^2C et à la caméra violerait ce principe. Nous nous restreindrons donc dans nos exemples, même si le langage autoriserait le contraire, à l'injection dans un unique conteneur.

Diagramme	Description
Diagramme d' implémentation	Permet d'afficher la totalité du modèle d'analyse de la plate-forme, avec ses ressources, périphériques, mondes et conteneurs et de glisser les objets dans les bons conteneurs.
Diagramme de comportement	Voir tableau 21 de la page 106.
Diagramme de trace	Voir tableau 21 de la page 106.

TABLE 28 – Diagrammes de la phase d'implémentation du système

8. En anglais, Single Responsibility Principle (SRP)

Exemples et notation. Le tableau 28 liste les trois types de diagramme utilisés durant la phase d'implémentation du système. Tout comme les phases précédentes, cette phase utilise les diagrammes de *comportement* et de *trace*, en plus d'ajouter un nouveau type de diagramme, le diagramme d'*implémentation*. Ce dernier permet de choisir les conteneurs dans lesquels seront implémentés les objets. Enfin, dans le cas particulier de cette phase, le diagramme de *comportement* n'est pas utilisé pour modéliser manuellement les comportements des objets, mais peut être consulté, après la ré-écriture automatique des machines à états pour vérifier la bonne construction de ces dernières, et des traces peuvent être à nouveau produites à partir des scénarios.

La figure 72 illustre le diagramme d'implémentation du système. Il offre une vue implémentée de tous les objets modélisés en conception du système dans les conteneurs de la plate-forme. Un objet d'implémentation dans un conteneur de la plate-forme est illustré par un rectangle à deux compartiments. Le nom du conteneur se trouve dans le premier compartiment, celui de l'objet implémenté dans le second. Les acteurs, ressources, périphériques, mondes et l'ensemble des associations conservent la même notation que lors des phases précédentes.

Les deux fragments du système ne dialoguent plus directement. La communication s'effectue désormais par l'intermédiaire des interfaces I^2C. Le fragment hébergé dans le monde Processeur a été injecté dans le conteneur d'accès à l'I^2C maître, ce qui lui permet d'initier une communication vers des interfaces I^2C esclaves. Tous les autres objets ont été implémentés dans les conteneurs correspondants permettant l'accès aux divers périphériques de la plate-forme. L'injection dans les conteneurs et l'exécution de règles de transformation a pour effet de modifier structurellement l'objet et de ré-écrire son comportement. Par exemple, l'injection de l'objet Détecteur de visage dans le conteneur d'accès à la caméra a eu pour effet d'ajouter une association entre l'objet et le périphérique Caméra. L'injection du fragment du système hébergé sur le processeur dans le conteneur d'accès à 'interface I^2C maître a eu pour effet de ré-écrire son comportement illustré par la figure 75b.

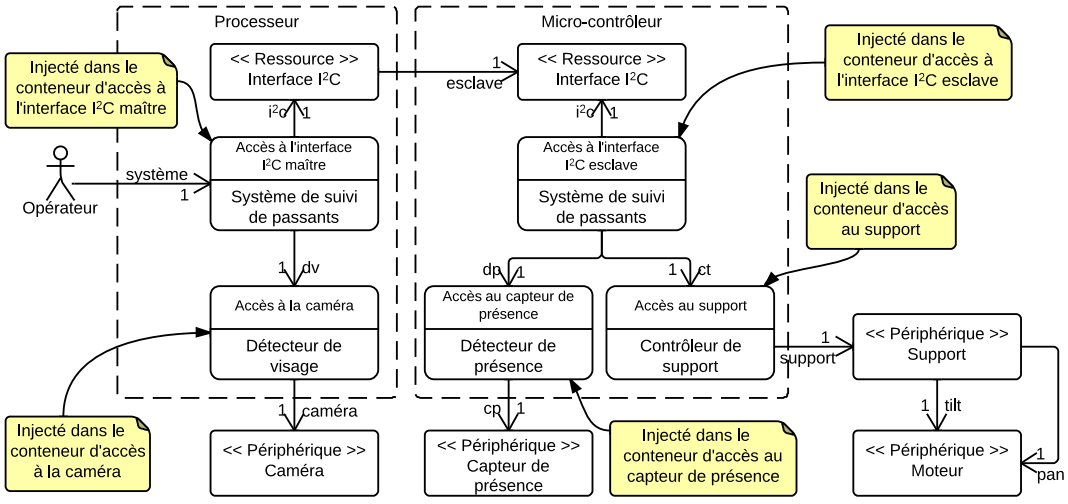


FIGURE 72 – Diagramme d'implémentation du système de suivi de passants

Id	Description
[1]	Les objets hébergés sur un monde de la plate-forme ne peuvent être injectés que dans les conteneurs fournis par ce monde.
[2]	Lorsqu'un objet injecté dans un conteneur est transformé par les règles d'implémentation du conteneur, celui-ci remplace ce dernier, toutes les associations issues du conteneurs vers les périphériques et les ressources sont également référencées par l'objet.

TABLE 29 – Liste des règles de cohérence intra-modèle pour l'implémentation du système

Règles de cohérence intra-modèle. Le méta-modèle d'implémentation étant un raffinement du méta-modèle de conception du système, les nouvelles règles proposées dans le tableau 29 viennent s'ajouter à celles déjà définies dans le tableau 26. La première s'assure qu'un objet hébergé dans un monde ne pourra bénéficier des règles d'implémentation des conteneurs fournies par d'autre monde. La seconde assure la cohérence du modèle après l'exécution des règles d'implémentation de l'objet.

Id	Description
[1]	À un objet du modèle de conception correspond le même objet dans le modèle d'implémentation.
[2]	À une association entre deux fragments objets hébergés dans le même monde dans le modèle de conception du système correspond la même association dans le modèle d'implémentation du système.

TABLE 30 – Liste des règles de cohérence inter-modèles pour l'implémentation du système

Règles de cohérence inter-modèles. Le tableau 30 liste les règles de cohérence à respecter afin de garantir que le modèle d'implémentation du système est valide vis-à-vis du modèle de conception obtenu dans la phase de *conception du système*. Une première règle permet de définir la correspondance entre les objets de conception et ceux d'implémentation. La seconde règle assure la correspondance entre les associations localisées dans les mondes. Cette correspondance n'a pas lieu entre les associations inter-mondes, ces dernières étant concrétisées par différentes implémentations (bus, NoC, etc.).

5.2 La démarche

La figure 73 illustre les différentes tâches durant la phase d'implémentation du système. Le point d'entrée de la phase d'implémentation sont les modèles de *conception du système* et d'*analyse de la plate-forme*.

Initialisation de la phase. La première étape consiste à initialiser le modèle d'implémentation en important les mondes de la plate-forme et les objets implémentés du *modèle de conception de l'application*. Durant l'initialisation, la description complète de la plate-forme est importée du modèle d'analyse de la plate-forme. Les éléments importés sont les ressources et les périphériques de la plate-forme, ainsi que les conteneurs – munis de leurs règles d'implémentation – dans lesquels seront implémentés les objets applicatifs provenant du modèle de

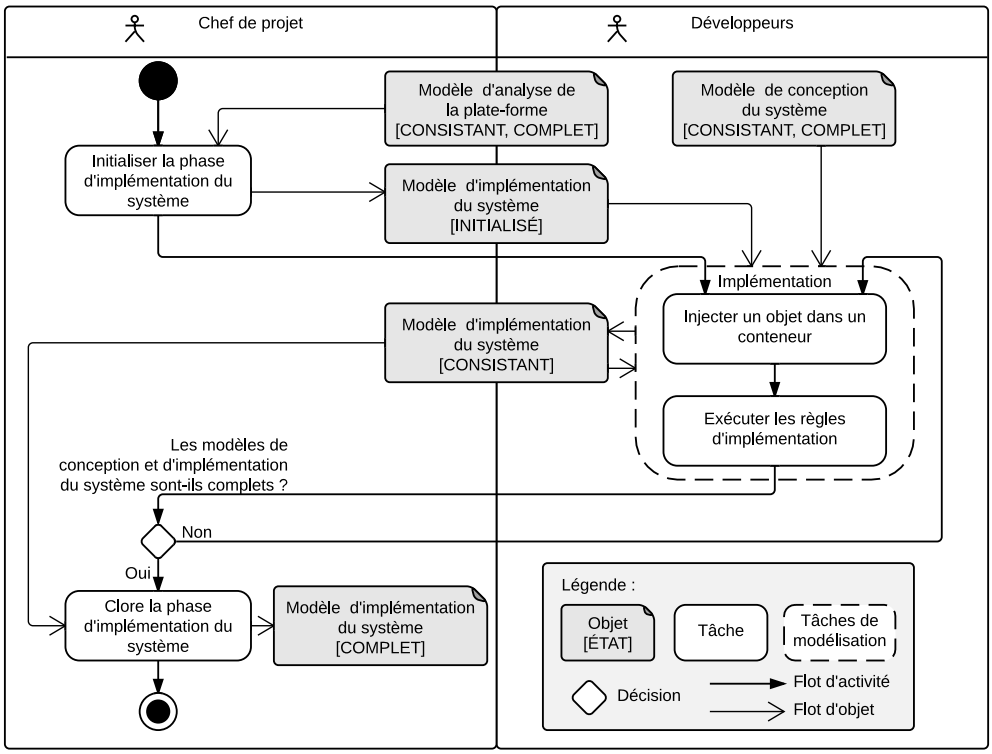
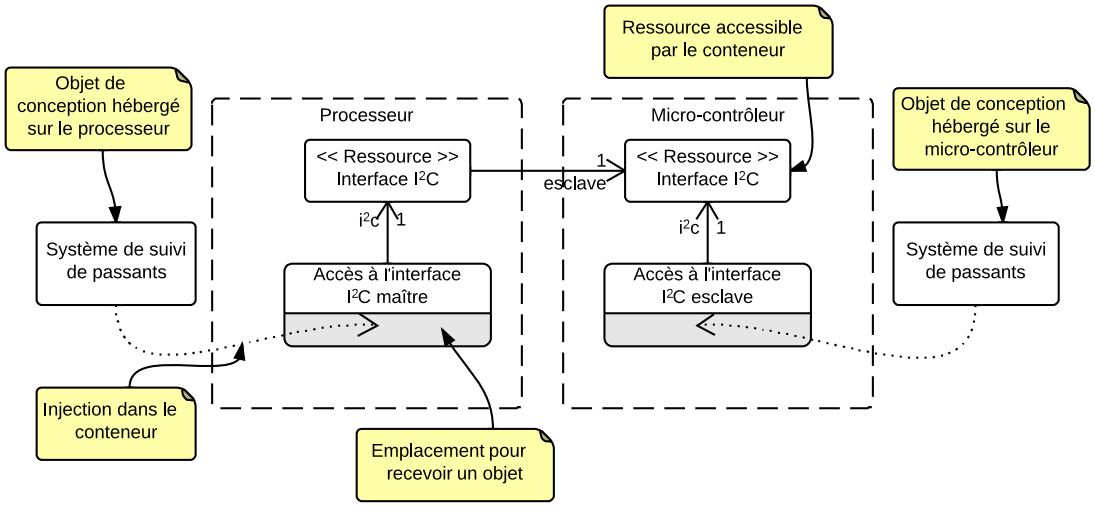


FIGURE 73 – Diagramme d'activités modélisant l'activité d'implémentation

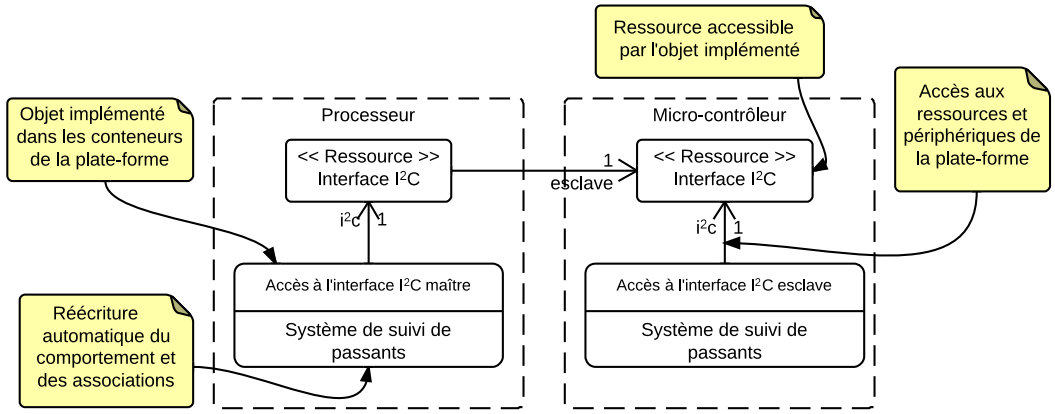
conception du système. Un modèle d'*implémentation du système* est alors INITIALISÉ et permet d'entamer la tâche de modélisation de la phase d'implémentation du système.

Tâches de modélisation « *Implémentation* ». La première tâche durant l'*implémentation* est l'*injection dans les conteneurs*. Elle permet au développeur de sélectionner les conteneurs de la plate-forme dans lesquels un objet est *injecté*. Ce choix est manuel et répond à des critères d'implémentation des objets applicatifs du système. Le développeur peut aussi pouvoir choisir entre des conteneurs différents, offrant des règles d'implémentation similaires mais n'utilisant pas les mêmes ressources ou périphériques (par exemple, l'implémentation d'un protocole de communication dans deux bus différents) et dans ce cas, il peut faire le choix de l'un ou l'autre, en fonction par exemple de contraintes d'implémentation (rapidité du bus, latence, perte de données, etc.). Enfin, il peut choisir d'injecter un objet dans plusieurs conteneurs qui offrent des services différents (par exemple, un conteneur pour piloter des moteurs et un conteneur pour utiliser une ressource de communication).

La figure 75 illustre l'injection des deux fragments du système qui ont été obtenus durant une activité de distribution en *conception du système* dans les conteneurs d'accès aux interfaces I^2C maître et esclave. Les deux conteneurs ont respectivement accès à l'interface de communication I^2C offerte par le monde dans lequel ils sont définis. La partie haute illustre l'activité manuelle d'injection dans les conteneurs, durant laquelle le conteneur est sélectionné, et l'objet est « déplacé » dans le compartiment de ce dernier. La partie basse de l'image illustre l'acti-



(a) Fragments du système hébergés sur le processeur (à gauche) et le micro-contrôleur (à droite) avant injection dans les conteneurs

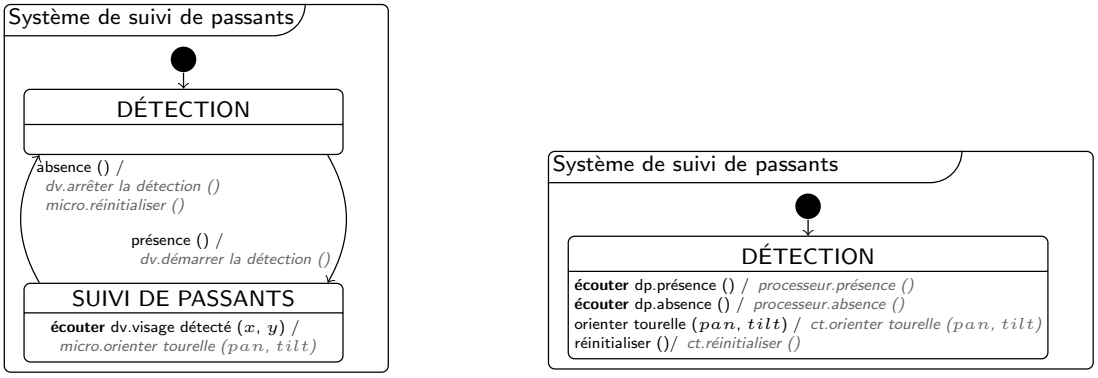


(b) Fragments du système hébergés sur le processeur (à gauche) et le micro-contrôleur (à droite) après l'exécution des règles d'implémentation

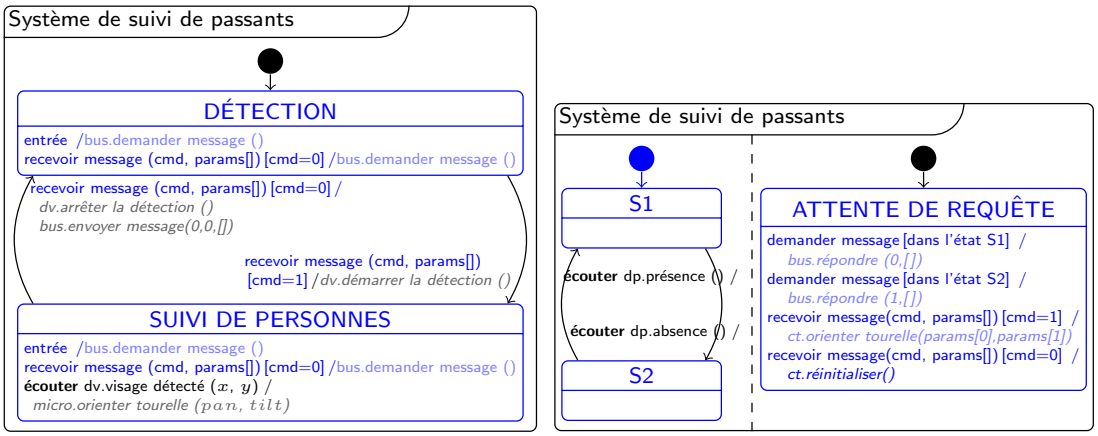
FIGURE 74 – Implémentation des objets après injection dans les conteneurs

vité automatique de ré-écriture des associations entre les objets. On peut également noter que l'association qui liait les deux fragments du système a été concrétisée par l'association entre les deux ressources de communication des deux mondes.

La figure 75 illustre la ré-écriture des comportements des objets implémentés durant l'activité automatique d'exécution des règles de transformation des conteneurs. La partie haute de la figure illustre les comportements des deux fragments du système avant leur conception tandis que la partie basse illustre la ré-écriture de ces comportements pour les objets implémentés en tenant compte des nouvelles associations entre les objets et les périphériques. Les modifications ont été illustrées en couleur bleu. La transformation majeure a été réalisée sur le fragment hé-



(a) Comportement des fragments du système hébergés sur le processeur (à gauche) et le micro-contrôleur (à droite) avant injection dans les conteneurs



(b) Comportements implémentés des fragments du système hébergés sur le processeur (à gauche) et le micro-contrôleur (à droite) après l'exécution des règles d'implémentation

FIGURE 75 – Implémentation du comportement des objets après injection dans les conteneurs

bergé dans le micro-contrôleur. Ce dernier ne pouvant initier une communication au travers du bus I^2C (en mode ESCLAVE), il ne peut plus directement prévenir le processeur d'une détection de présence ou d'absence. De ce fait, deux nouveaux états S1 et S2 [9] permettent de vérifier l'état de la détection. Dans l'état DÉTECTION, lorsque le fragment du système hébergé sur le processeur initialise une conversation (avec l'action *i2c.demander message()*) et attend une réponse du micro-contrôleur (avec l'événement *recevoir message (cmd, params[])*). Le fragment du système sur le micro-contrôleur est en ATTENDE DE REQUÊTE. En la recevant, il peut alors envoyer son état (non détection ou détection, induit de S1 ou S2) qui est *sérialisé* et envoyé au travers du bus. Le premier système attend le retour de l'esclave et en extrait la commande

9. Ces noms sont arbitraires. Les machines à états ont été ré-écrites grâce aux règles de transformation de code du conteneur d'accès à l'interface I^2C esclave. Ces règles ne peuvent deviner quelle est la signification de ces états et assignent donc des noms arbitrairement.

ainsi que les éventuels paramètres.

Clôture de la phase. La clôture de la phase n'est possible que lorsque tous les modèles des phases amont sont considérés *complets* et tous les scénarios doivent pouvoir être rejoués afin de vérifier que tout le besoin est satisfait. Dans ce cas-là, il est possible de clore cette phase ce qui a pour effet de faire passer le modèle d'*implémentation du système* dans l'état COMPLET.

EN RÉSUMÉ

- ✓ *Activité assurant l'implémentation d'applications dans les plates-formes*
- ✓ *Intégration complète du modèle d'analyse de la plate-forme*
- ✓ *Activité d'injection des objets applicatifs dans les conteneurs de la plate-forme*
- ✓ *Activité automatique de ré-écriture des objets (associations et comportement)*
- ✓ *Injection dans les conteneurs permettant d'assurer l'interfaçage des objets applicatifs avec les périphériques et les ressources de la plate-forme*

Synthèse du chapitre

Ce chapitre a permis de montrer plusieurs aspects de la méthode ⟨HOE⟩² et notamment son processus et son langage. Le langage commun permet de modéliser des systèmes embarqués du côté de l'application et de la plate-forme. La formalisation stricte des activités du processus, notamment en termes de produits (et d'états des produits) en entrée nous permettent de mettre en évidence les bénéfices apportés par le processus. Nous avons proposé une logique dans l'établissement des états des modèles, tout d'abord INITIALISÉ, ensuite CONSISTANT et enfin COMPLET. Cet enchainement qui se veut simple et commun à tous les modèles du processus assure le suivi du processus, guidé par une largeur du besoin, mettant en lumière l'aspect incrémental de l'approche et favorise un développement rapide et efficace.

La démarche a été formalisée dans la même logique de simplicité et de consistance. Nous avons distingué les tâches d'initialisation et de clôture de chacune des phases des tâches de modélisation. Les aspects décisionnels, permettant de décider quand les phases doivent être initialisées ou closes, ainsi que dans quel ordre doivent être réalisées les tâches de modélisation relèvent du chef de projet, acteur actif dans le processus ⟨HOE⟩².

Le flot proposé ne prend pas en compte la composition de plates-formes énoncée dans le chapitre précédent. Dans le prochain chapitre, nous avons étendu la formalisation de ce processus et du langage afin de prendre en compte cet aspect.

Chapitre 7

Systèmes embarqués et composition de plates-formes

Publication pertinente à ce chapitre :

[**Hili *et al.* 2014b**] Nicolas Hili, Christian Fabre, Sophie Dupuy-Chessa, Dominique Rieu et Ivan Llopard. *Model-Based Platform Composition for Embedded System Design.* In IEEE 8th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC-14) (IEEE MCSoC-14), Aizu-Wakamatsu, Japan, Septembre 2014

Liste des sections

Dans ce chapitre, nous présentons notre contribution sur la composition de plates-formes. Nous appelons composition de plates-formes le fait de pouvoir construire une plate-forme à partir d'une ou plusieurs autres. Tout l'enjeu de la composition est d'évaluer et de modéliser l'impact sur le code applicatif qui doit être hébergé sur les différents morceaux de la plate-forme.

La composition de plates-formes s'inscrit sur deux axes. Horizontalement, où des plates-formes de même niveau d'abstraction peuvent être *assemblées*, nous appelons alors ce type de composition l'*assemblage* de plates-formes, et verticalement, où une plate-forme est conçue sur la base d'une autre et offre plus de services ou des services plus évolués que cette dernière, nous appelons ce type de composition l'*incrément* de plates-formes.

Nous proposons dans ce chapitre un modèle de composition de plates-formes permettant d'unifier les deux types de composition et montrons comment les concepts de *monde* et de *conteneur* sont suffisants pour maitriser l'impact de la composition sur le code applicatif généré. Pour ce faire, nous avons pris en compte la composition de plates-formes au niveau du processus et du langage de la méthode. Nous avons présenté la composition, de façon indépendante, du

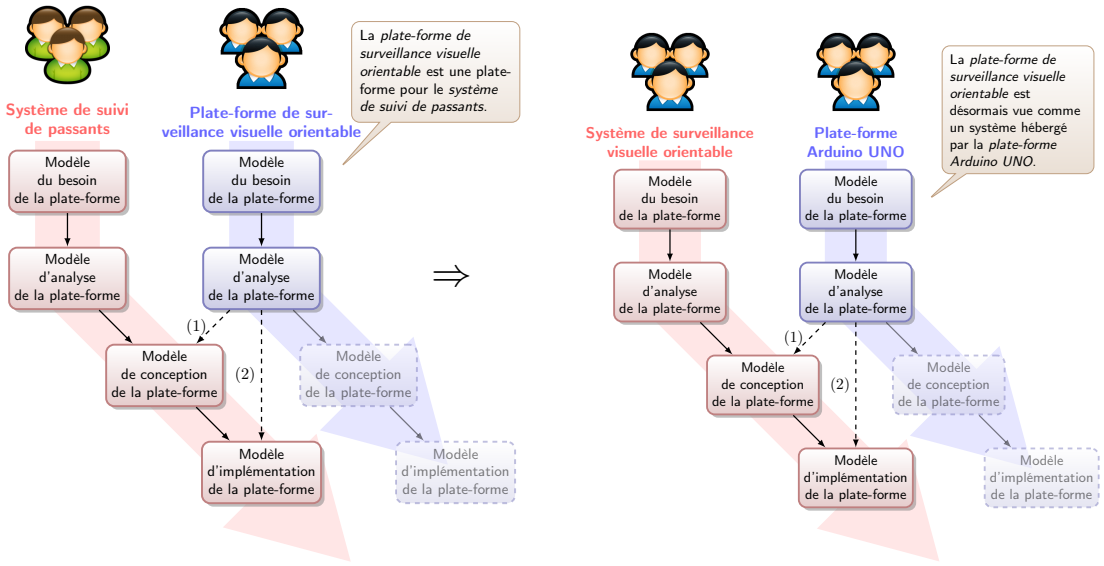
point de vue processus dans [Hili 2013b] et du point de vue langage dans [Hili *et al.* 2014b]. Ce chapitre présente la composition de la façon suivante : la section 1 introduit la composition et la manière dont nous la prenons en compte dans ⟨HOE⟩^{2} ; la section 2 présente notre modèle de composition de plates-formes.

1 Introduction à la composition de plates-formes

Afin d'introduire notre modèle de composition de plates-formes, cette section est structurée en trois parties. Les deux premières abordent la composition des points de vue du processus et du langage ; la dernière introduit certaines considérations sur la manière de composer.

1.1 Composition au sein du processus ⟨HOE⟩^{2}

Conceptuellement, *système*, *application* et *plate-forme* sont des notions relatives entre elles. Si un système peut être considéré comme étant une plate-forme vis-à-vis d'une application qu'il est en mesure d'héberger et à qui il peut offrir des services (accès aux ressources et aux périphériques), il peut également être considéré comme une *surcouche applicative* implémentée sur une autre plate-forme. Application et plate-forme constituent donc des points de vue relatifs entre eux. Afin d'illustrer ce changement de point de vue, considérons le *système de suivi de passants* et sa *plate-forme de surveillance visuelle orientable* (cf. fig. 76a). Au vu de sa plate-forme, les objets modélisés sont des objets applicatifs s'implémentant sur les ressources et périphériques de la plate-forme. Par exemple, l'objet applicatif Détection de présence accède au périphérique Capteur de présence de la plate-forme. Considérons maintenant que cette plate-



(a) Système de suivi de passants hébergé par la plate-forme de surveillance visuelle orientable

(b) plate-forme de surveillance visuelle orientable hébergée par la plate-forme Arduino UNO

FIGURE 76 – Composition au sein du processus ⟨HOE⟩^{2}

forme est de son côté implémentée sur une plate-forme plus *primitive*, telle que la plate-forme *Arduino UNO* (cf. fig. 76b). Le périphérique Capteur de présence de la *plate-forme de surveillance visuelle orientable* est une surcouche du périphérique Broche de la plate-forme *Arduino UNO*. Vis-à-vis de cette dernière plate-forme, le périphérique Capteur de présence est perçu comme un objet offrant des services de plus haut niveau (détection de présence ou d'absence) que le périphérique Broche (changement d'état), lui-même étant une abstraction d'un objet offrant des services plus primitifs (changement de polarisation du transistor de sortie interne à la broche). Ce changement de point de vue introduit la composition de plates-formes.

Pour prendre en compte la composition de plates-formes, nous avons étendu le processus $\langle \mathsf{HOE} \rangle^2$ en considérant désormais plusieurs branches pour le développement d'une plate-forme composée. Chaque branche illustre le développement d'une plate-forme pour la branche à sa gauche et celui d'une surcouche applicative de la branche de droite. Lorsqu'une plate-forme s'appuie à son tour sur une ou plusieurs autres, il est possible de raffiner son développement sur les phases de *conception* et d'*implémentation du système*. La *plate-forme de surveillance visuelle orientable* est représentée en rouge sur la figure 76b [1] puisqu'elle joue le rôle de *surcouche applicative* sur la plate-forme Arduino UNO. Cette dernière peut également être modélisée sur les bases d'une autre plate-forme et ainsi de suite. Ainsi, le processus propose une structure récursive afin d'itérer sur les différents niveaux de plates-formes.

1.2 Composition au sein du langage $\langle \mathsf{HOE} \rangle^2$

Les figures 77a et 77b présentent deux vues du méta-modèle de *composition du système*. Ce dernier raffine le méta-modèle d'*implémentation du système*. Sur les deux figures, nous avons distingué les concepts propres à la modélisation d'un système (en rouge) de ceux propres à la modélisation des plates-formes (en bleu) et avons mis en surbrillance les concepts et les relations nécessaires à la compréhension de la vue. La figure 77a illustre la vue « *analyse* » d'un système. Les différentes relations permettent la modélisation du système pouvant être une application ou une plate-forme. La figure 77b illustre la vue « *implémentation* » de la partie *applicative* du système *sur* la plate-forme du système.

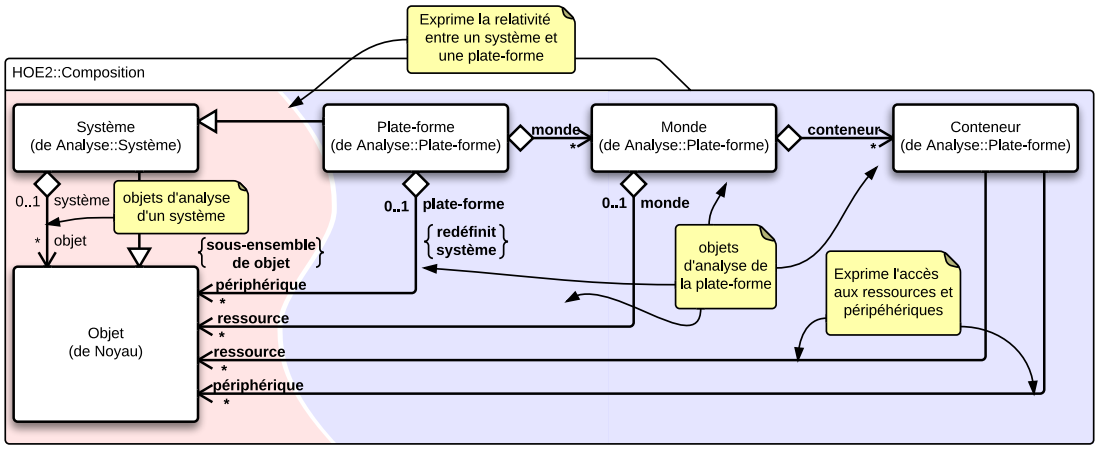
Afin de composer des plates-formes, nous avons renommé la relation *hébergé par* entre les concepts *Système* et *Plate-forme* du méta-modèle d'*implémentation du système* de la figure 71 de la page 122, par la relation *compose* du méta-modèle de la figure 77. La cardinalité « *0..1* » a été remplacée par la cardinalité « * », permettant de modéliser un système hébergé sur plusieurs plates-formes, comme c'est le cas avec la *plate-forme de surveillance visuelle orientable* hébergée sur la *plate-forme Arduino UNO*.

La figure 78 illustre l'instanciation du méta-modèle de *composition du système* pour le *système de suivi de passants* Le système est hébergé sur la *plate-forme de surveillance visuelle orientable*. Cette dernière est une plate-forme mais également un système pouvant composer d'autres plates-formes. Elle compose notamment la *plate-forme Arduino UNO*.

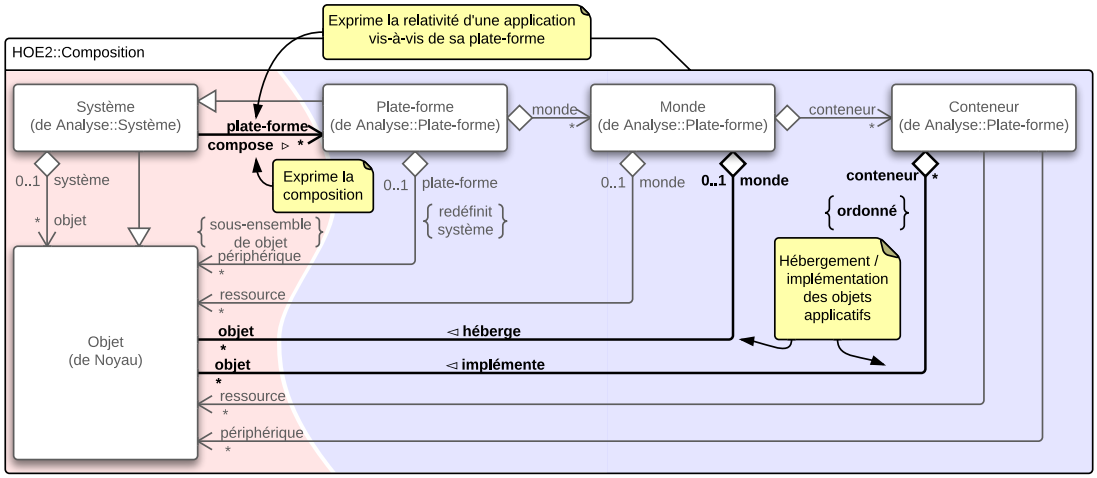
1.3 Considérations sur le choix et le nombre de compositions.

Nous avons montré dans la partie précédente que d'un point de vue conceptuel, *système*, *application* et *plate-forme* sont des concepts relatifs. Un système peut à la fois être considéré

1. le terme « *plate-forme* » a été remplacé par « *système* » pour montrer son changement de rôle vis-à-vis de la *plate-forme Arduino UNO*.



(a) Vue « analyse » du système et de la plate-forme



(b) Vue « implémentation » sur la plate-forme

FIGURE 77 – Méta-modèle de composition des plates-formes

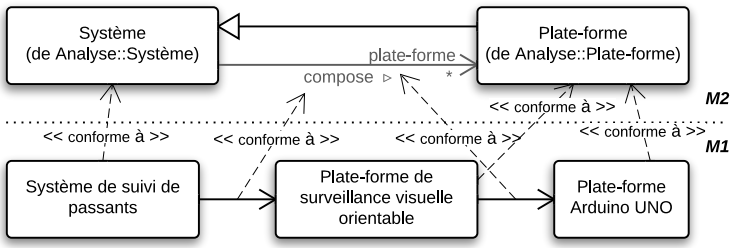


FIGURE 78 – Instanciation du méta-modèle

comme une plate-forme pour ses applications et comme une couche applicative pour sa propre plate-forme. De là prend racine la composition de plates-formes. La composition est un moyen

récursif d'organiser des plates-formes les unes par rapport aux autres. La composition de plates-formes présente des bénéfices majeurs en termes d'organisation des équipes de développement et de parallélisation des développements des sous-ensembles d'un système constitué de plusieurs niveaux de plates-formes. En séparant le développement sur plusieurs flots indépendants, il est théoriquement possible de profiter pleinement de la parallélisation des développements et ainsi réduire le temps de développement, en dehors de toute considération de la réutilisation d'un niveau de plates-formes pour la conception de différents systèmes.

Du point de vue pratique, si le nombre de plates-formes composées est trop grand et inadapté à la taille de l'équipe, la composition risque d'alourdir le développement. Tout l'enjeu est alors de définir ce qui constitue effectivement une plate-forme et combien de niveaux de plates-formes il peut être envisagé de composer afin de bénéficier pleinement de la composition. Ce choix doit être fait au regard des fonctionnalités que doivent fournir les différentes plates-formes composées. En cela, le choix de l'organisation de différents niveaux de plates-formes peut être dicté par la modélisation du modèle d'analyse du besoin de chacune d'elle. Le processus $\langle \mathsf{HOE} \rangle^2$ est cohérent avec cette approche dictée par l'analyse du besoin. La figure 79 constitue une vue dite « *de dessus* » qui illustre l'organisation des plates-formes que nous choisissons désormais pour le développement du *système de suivi de passants* et ce pour tout le reste de ce chapitre. Elle correspond à un plan de coupe d'une vue « processus » (les figures 76a et 76b sont des exemples de vues « *processus* ») dans laquelle nous ne nous intéressons qu'à représenter les relations entre les différentes plates-formes, les plates-formes les plus à droite étant considérées comme *finales*.

La *plate-forme finale* est une considération purement arbitraire. Elle définit la plate-forme à partir de laquelle on considère que le niveau de détail est suffisant pour obtenir une implémentation du système sur cette dernière. Durant nos travaux, nous nous sommes concentrés sur de la génération de code de haut niveau (C, C++, Python) pour des plates-formes mono et multi-processeurs embarqués. Cependant, la façon de composer n'exclut toutefois pas la possibilité de descendre à des niveaux de plates-formes et des niveaux d'implémentation beaucoup plus primitifs (assembleur, VHDL, ASIC, etc.). Là encore, l'estimation de ce qu'est la plate-forme finale correspond à des décisions d'ingénierie prises en fonction du projet et des degrés de précision et d'optimisation voulus. La génération de circuit ASIC est beaucoup plus spécifique que la génération de code C pour une plate-forme Linux mais introduit plus de niveaux d'implémentation, des modèles de plates-formes complexes dans lesquels de fortes règles d'implémentation et contraintes physiques et temporelles sont injectées.

Dans le cas du *système de suivi de passants* développé dans ces travaux, nous avons considéré les plates-formes Arduino UNO [Arduino Official Website 2014] et Raspberry PI [Raspberry PI Foundation 2014] comme finales et avons généré du code sur ces dernières. Notre contribution a permis d'identifier deux types de composition possibles pour la conception de systèmes embarqués. Ces deux compositions sont illustrées sur la figure 79. Elles ne nomment respectivement l'*incrément* et l'*assemblage*.

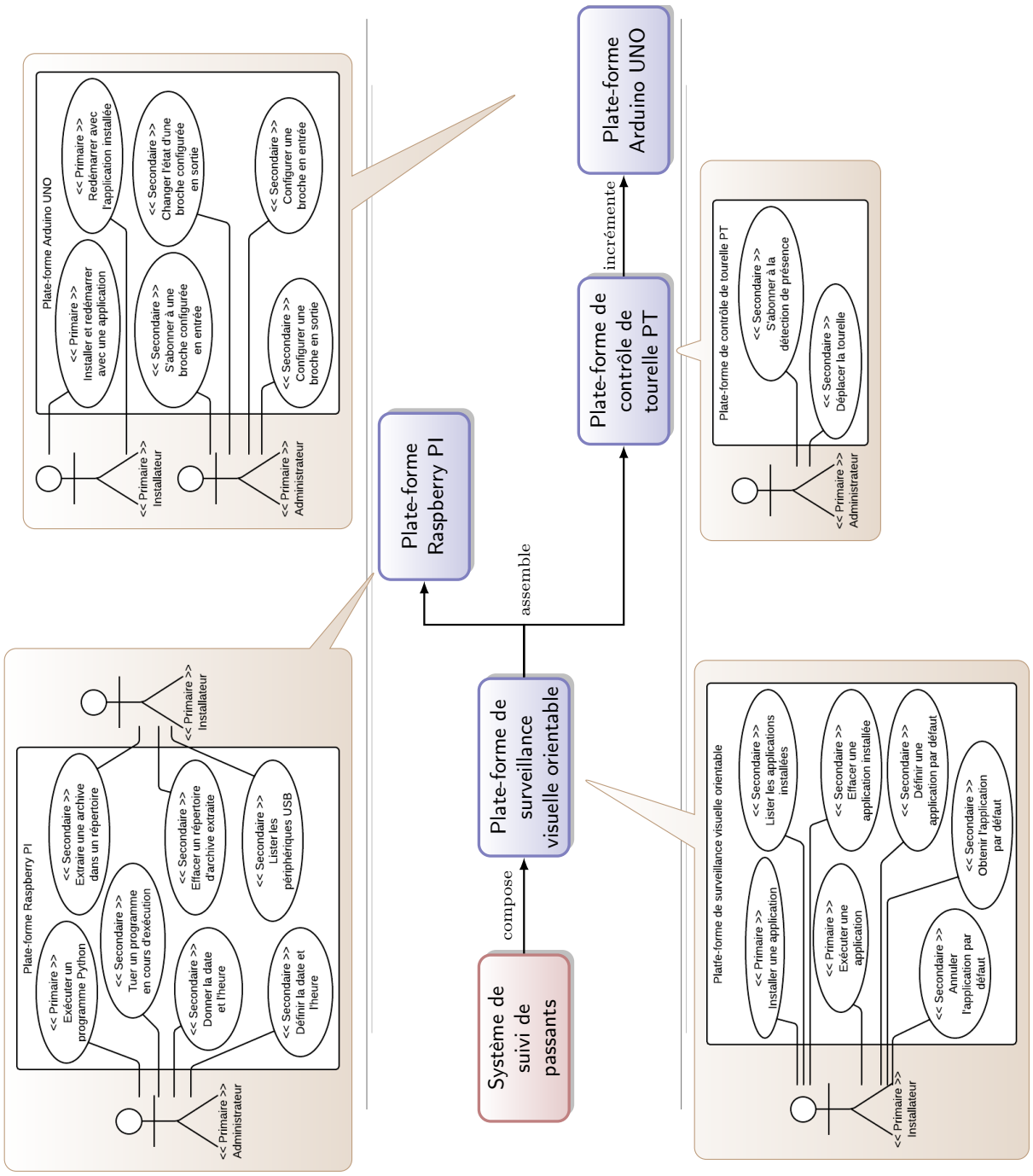


FIGURE 79 – Composition de la plate-forme de surveillance visuelle orientable

EN RÉSUMÉ

- ✓ *Application et plate-forme sont des points de vues relatifs d'un système*
- ✓ *Un système peut donc jouer le rôle d'application pour la plate-forme qui l'héberge et de plate-forme pour une autre application qu'il héberge*
- ✓ *Le processus $\langle HOE \rangle^2$ offre un moyen de composer les plates-formes de manière récursive*
- ✓ *La plate-forme finale est la plate-forme à partir de laquelle on génère le code applicatif*

2 Modèle de composition de plates-formes

Dans la suite de ce chapitre, nous présenterons notre modèle de composition de plates-formes appliqué sur le cas d'étude du *système de suivi de passants*. Avant de présenter indépendamment les deux types de composition et ensuite le regroupement des deux, la première partie de cette section présente les différents niveaux de plates-formes de la figure 79 de la page 134.

2.1 Présentation des différents niveaux de plates-formes

La *plate-forme de surveillance visuelle orientable* illustrée dans la figure 79 est la plate-forme de plus haut niveau sur laquelle nous développerons les modèles applicatifs du *système de suivi de passants*. Cette plate-forme possède des cas d'utilisation d'installation et d'exécution d'applications et peut lister les applications installées. Enfin, elle peut définir, obtenir et annuler l'application par défaut. Lorsque la plate-forme est redémarrée, dans le cas où elle possède une application par défaut, elle l'exécute immédiatement.

La *plate-forme Raspberry PI* permet l'installation et la désinstallation de programmes Python et l'exécution d'un seul programme à la fois. Elle est adaptée au domaine du traitement d'image, et offre des cas d'utilisation de démarrage et d'arrêt d'une *caméra*. D'autres cas d'utilisation (affichage et modification de la date et de l'heure) sont également modélisés.

La *plate-forme de contrôle de tourelle PT* offre quant à elle des cas d'utilisation pour s'abonner à la détection de présence, et déplacer une tourelle PT (Pan & $Tilt$). Elle est adaptée au domaine de la surveillance. La tourelle PT peut être utilisée pour supporter des capteurs ou des actionneurs, tels qu'une caméra, un radar ultrason, un projecteur, etc.

La *plate-forme Arduino UNO* est une plate-forme de prototypage dédiée au contrôle de capteurs et d'actionneurs numériques et analogiques. Cette dernière est en mesure de recevoir une seule application à la fois. Elle possède un cas d'utilisation pour l'installation d'une application mais pas pour la désinstallation, l'application étant automatiquement écrasée dès qu'une nouvelle application est installée. En outre, étant une plate-forme de prototypage, elle offre des cas d'utilisation spécifiques à la configuration et l'utilisation de broches permettant d'interfacer différents capteurs et actionneurs (moteurs, leds, sondes de température, etc.).

Du point de vue de la composition, les deux plates-formes finales sont les plates-formes Arduino UNO et Raspberry PI. La première possède un micro-contrôleur simple. Son développement s'effectue dans le langage *Arduino*, qui est un dérivé du langage C++. Elle est cependant très limitée et insuffisante pour du traitement d'image. La plate-forme Raspberry PI est un micro-ordinateur embarqué supportant le système d'exploitation Linux. Elle incorpore la librairie OpenCV pour le traitement vidéo. Elle possède également quelques broches, mais en nombre insuffisant pour pouvoir gérer tous les capteurs et actionneurs du système. Cette plate-forme est adaptée aux langages de développement C et Python. Dans la suite, nous ne considérerons qu'un fragment suffisant de ces plates-formes pour illustrer la composition de plates-formes.

2.2 Composition par incrément

La composition *par incrément* consiste à construire une plate-forme plus évoluée à partir d'une autre plate-forme plus primitive. Par commodité de langage, nous appellerons *plate-forme incrémentée* la plate-forme issue de l'incrément d'une autre plate-forme et *plate-forme primitive* la plate-forme à partir de laquelle est réalisé l'incrément. Du point de vue de la conception, un incrément définit une nouvelle plate-forme permettant d'offrir de nouveaux services, de simplifier l'usage de services existants ou bien de restreindre ou interdire l'utilisation des services de la plate-forme primitive. Ainsi, il est plus facile de concevoir un système s'appuyant sur des services évolués d'une plate-forme plutôt que sur les services primitifs d'une autre.

Afin d'illustrer l'incrément pour le développement du *système de suivi de passants*, nous considérons dans cette partie deux plates-formes, la *plate-forme de contrôle de tourelle PT*, et la *plate-forme Arduino UNO*. La *plate-forme de contrôle de tourelle PT* est la plate-forme *incrémentée* sur la *plate-forme Arduino UNO*, comme le suggère la figure 79.

La composition par incrément au sein du processus. La figure 80 illustre la composition *par incrément* de plates-formes au sein du processus. La branche la plus à gauche représente le développement du système, la branche intermédiaire celui de la plate-forme incrémentée et la branche la plus à droite celui de la plate-forme primitive. Dans le cadre de cet incrément, la *plate-forme Arduino UNO* est la plate-forme primitive, la plate-forme incrémentée se nomme *plate-forme de contrôle de tourelle PT*. Un plan de coupe est réalisé sur la figure et la vue coupée illustre l'agencement des plates-formes entre elles. Du point de vue de l'ingénierie des systèmes embarqués, un tel processus confère de nombreux avantages. Il permet de diviser l'équipe de développement sur les différents flots qui sont selon le processus ⟨HOE⟩[2] indépendants. Cela permet de modéliser des fonctionnalités de la plate-forme en parallèle et ainsi de réduire le temps de développement. Cette organisation facilite également la tâche des développeurs des plates-formes qui n'ont à traiter que des problématiques locales à un niveau d'abstraction.

Le modèle d'*implémentation du système de surveillance visuelle orientable* n'est plus obtenu en une fois mais est lui-même progressivement raffiné. La première implémentation du système repose sur la définition du modèle d'*analyse de la plate-forme incrémentée*. Le modèle de cette dernière étant lui-même raffiné durant les phases de *conception* et d'*implémentation du système*, ce raffinement se répercute sur le modèle d'*implémentation du système de surveillance visuelle orientable* qui peut être *automatiquement* raffiné pour atteindre une implémentation sur la *plate-forme Arduino UNO*.

Le raffinement automatique de ce système sur la *plate-forme Arduino UNO* est rendu possible *par transitivité* au regard de la tâche d'injection dans les conteneurs durant la phase d'implémentation du système. La figure 81 illustre cette transitivité. En considérant que le *conteneur d'accès au capteur de présence* de la *plate-forme de contrôle de tourelle PT* est injecté selon la tâche d'*injection* dans le *conteneur d'accès à une broche* de la *plate-forme Arduino UNO* (fig. 81a), alors l'objet Détecteur de présence du système injecté dans le *conteneur d'accès au détecteur de présence PIR*[2] est par transitivité également injecté dans le *conteneur d'accès à une broche* (fig. 81d). De manière plus générale, la première injection produit un objet d'implémentation sur la *plate-forme de contrôle de tourelle PT* (fig. 81b), ce dernier étant lui-même implémenté dans la *plate-forme Arduino UNO* (fig. 81c).

2. En anglais, PIR.

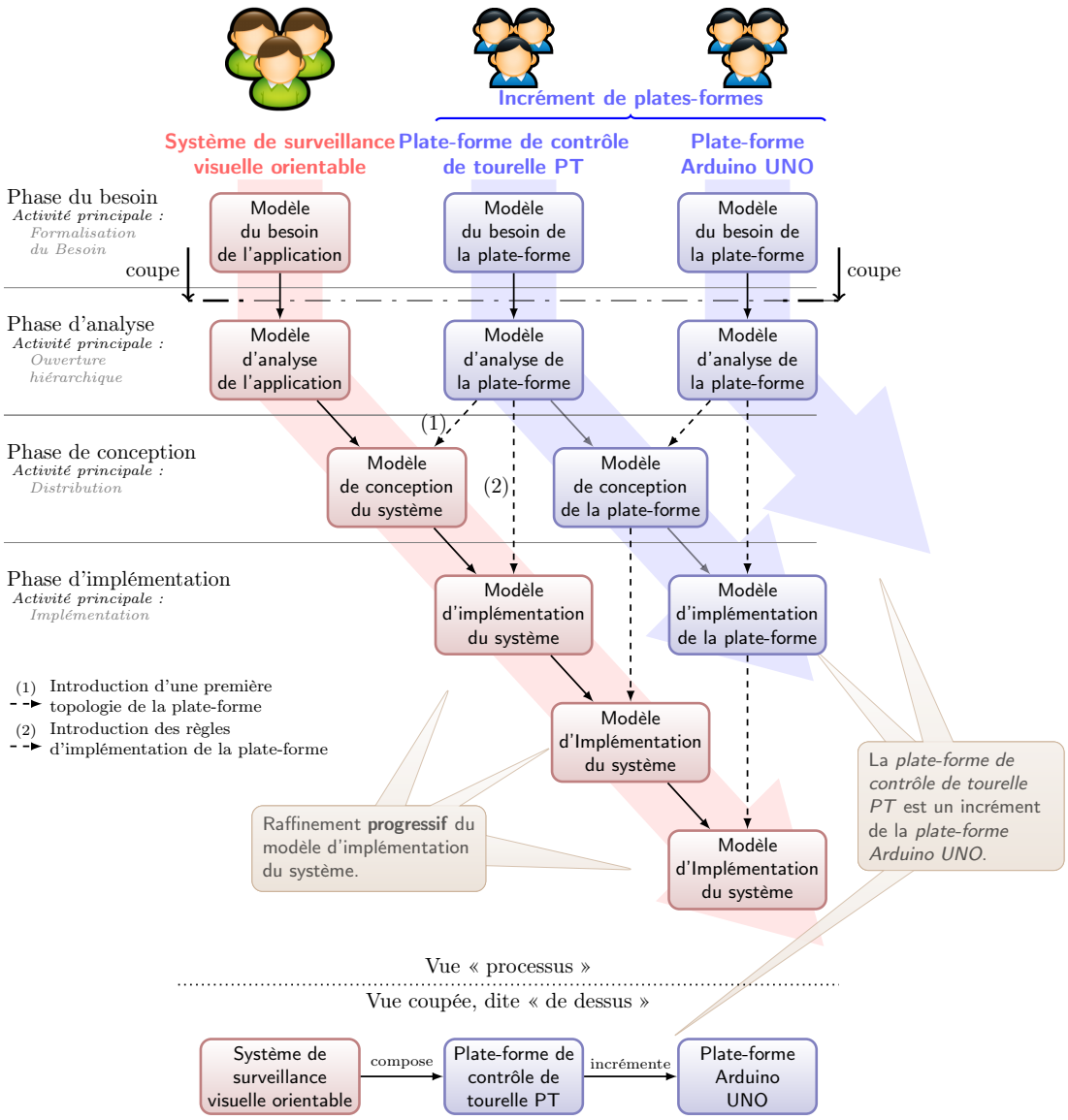
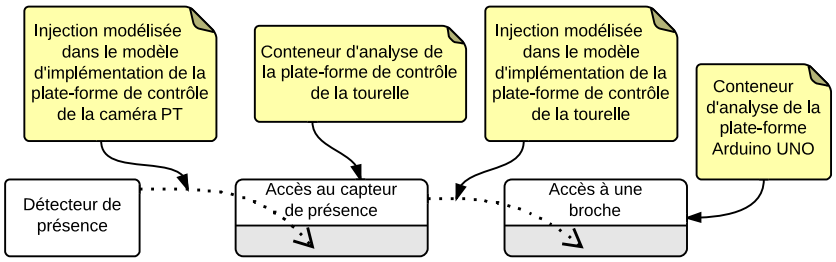


FIGURE 80 – Composition par incrément

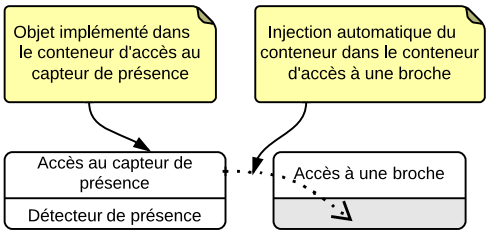
Plus formellement, l'injection dans les conteneurs peut se noter par une relation transitive binaire $\mathcal{R}$. Ainsi, en considérant un conteneur $c_1 \in \mathbb{O}$ de la plate-forme incrémentée et $c_2 \in \mathbb{O}$ de la plate-forme primitive, et $o \in \mathbb{O}$ un objet du système à implémenter sur la plate-forme incrémentée, l'équation 7.1 traduit la transitivité de l'injection dans les conteneurs.

$$\forall o, c_1, c_2 \in \mathbb{O}, o\mathcal{R}c_1 \wedge c_1\mathcal{R}c_2 \implies o\mathcal{R}c_2 \tag{7.1}$$

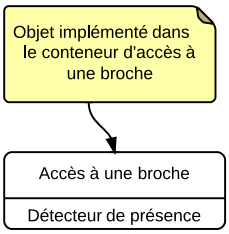
Cette transitivité est possible au sein du langage $\langle \mathsf{HOE} \rangle^2$ car le conteneur est un objet spécial d'une plate-forme. À ce titre, il étend le concept d'objet et peut donc être lui-même injecté



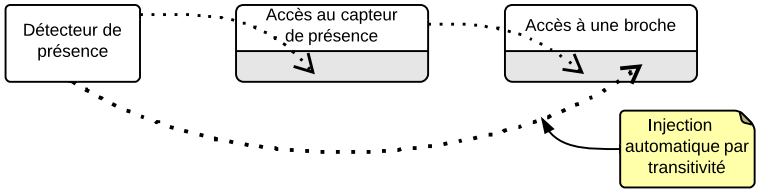
(a) Injection multiple sur plusieurs plate-formes par incrément



(b) Première implémentation



(c) Seconde implémentation



(d) Transitivité de l'injection dans les conteneurs

FIGURE 81 – Injection successive dans les conteneurs

dans un autre conteneur. La transitivité est assurée au niveau des règles d'implémentation des conteneurs (qui seront approfondies dans le chapitre 10). Elle apporte un gain en termes de réécriture automatique de machines à états et de réduction des activités manuelles de modélisation de comportement.

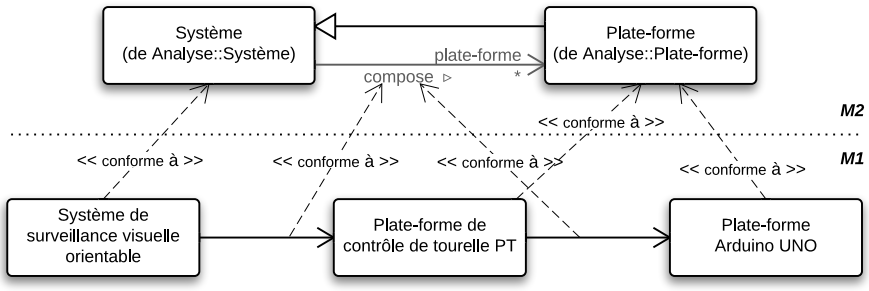


FIGURE 82 – Composition par incrément au sein du langage

La composition par incrément au sein du langage langage. Du point de vue du langage, le méta-modèle présenté au début de ce chapitre permet d'organiser les plates-formes pour réaliser l'incrément. La figure 82 illustre la modélisation d'un diagramme d'objets conforme au méta-modèle. Dans ce modèle, le *système de surveillance visuelle orientable compose* une plate-forme, elle-même *composée* (par incrément) d'une seconde plate-forme. Nous n'avons pas défini de notation spéciale pour illustrer la composition de plates-formes. Une fois les plates-formes composées (ce choix s'effectue en prémisse de la phase de *conception du système*), les mondes de la ou des plates-formes sont introduits dans le flot de développement du système.

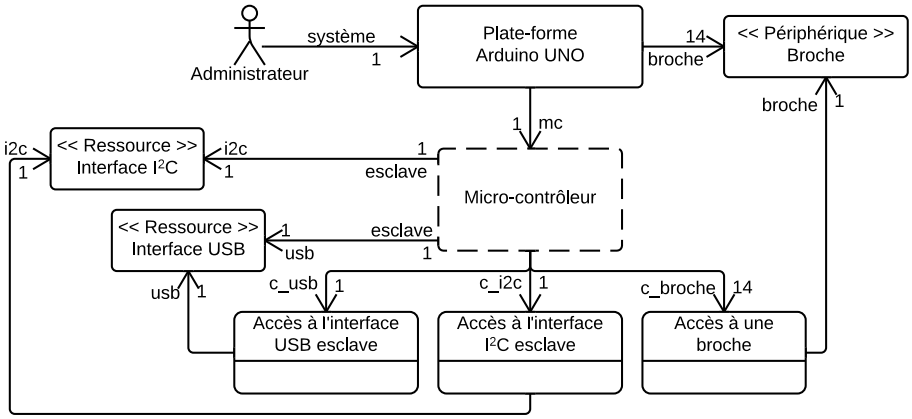


FIGURE 83 – Modèle d'analyse de la plate-forme Arduino UNO

Incrément de la plate-forme Arduino UNO. La figure 83 illustre le modèle d'*analyse de la plate-forme Arduino UNO*. Nous avons modélisé quatorze broches numériques et des interfaces d'accès aux bus I²C et USB permettant à la plate-forme Arduino UNO d'être connectée à d'autres plates-formes. La plate-forme modélisée offre un monde permettant d'héberger une application et trois types de conteneurs. Des conteneurs de broches, permettant l'accès et la manipulation des quatorze broches de la plate-forme, un conteneur pour l'accès au périphérique I²C et un dernier pour l'accès au périphérique USB.

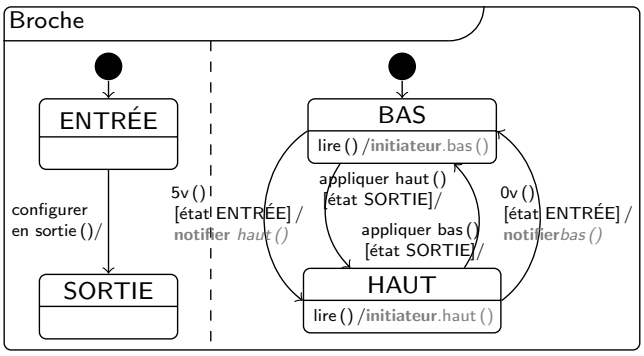


FIGURE 84 – Comportement de la broche de la plate-forme Arduino UNO

La figure 84 illustre le comportement de l'objet Broche. Une broche est initialement configurée en ENTRÉE (état haute impédance). Elle joue le rôle de *capteur binaire* et permet de mesurer des grandeurs physiques de l'environnement par l'intermédiaire de capteurs physiques, tels que des capteurs de proximité ou de contact. Elle possède deux valeurs d'états. L'état BAS signifie que le capteur ne détecte aucun signal [3] et l'état HAUT qu'un signal est détecté. En cas de commutation des états HAUT et BAS, la broche *notifie* son état aux autres objets du modèle qui l'*écoute*. Il est également possible à tous les objets du modèle de demander son état courant. La présence du mot-clé **initiateur** dans le modèle de comportement de la broche permet à la broche de retourner son état. La broche peut également être configurée en SORTIE, au moyen du message *configurer en sortie ()*. Dans ce cas-là, elle joue le rôle d'*actionneur* et permet le contrôle d'actionneurs physiques, tels que des servomoteurs ou des leds. Configurée en SORTIE, il est possible de commuter la broche dans l'état BAS ou HAUT.

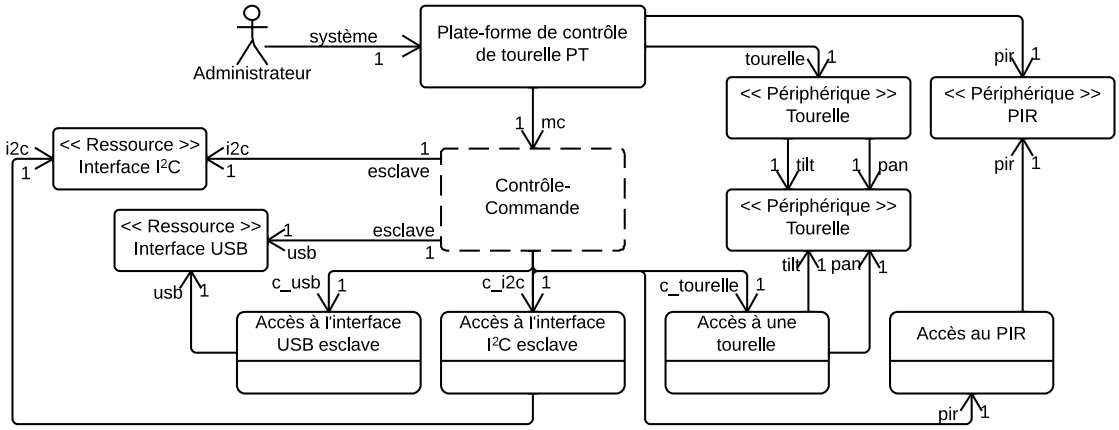


FIGURE 85 – Modèle d'analyse de la plate-forme de contrôle de tourelle PT

Les services offerts par la plate-forme par l'intermédiaire de ces conteneurs sont trop primitifs pour directement développer une application hébergée sur cette plate-forme. Pour faciliter leur utilisation et adapter la plate-forme au domaine de la surveillance, le choix a été fait de réaliser un *incrément* sur cette plate-forme. La plate-forme incrémentée se nomme *plate-forme de contrôle de tourelle PT*. Le modèle d'*analyse de la plate-forme* est illustré par la figure 85. Dans ce modèle, nous avons représenté un périphérique de détection de présence PIR et deux moteurs *pan* et *tilt* formant la tourelle PT. Cette plate-forme offre un monde pour héberger des objets et différents conteneurs. Les deux premiers conteneurs sont des conteneurs de communication aux travers des interfaces USB et I²C. Un troisième conteneur permet de contrôler la tourelle par l'intermédiaire des deux moteurs, et enfin le dernier permet de s'abonner aux PIR et ainsi d'être notifié en cas de détection de présence.

Dans le contexte de la composition *par incrément*, la *plate-forme de contrôle de tourelle PT* peut être raffinée en conception et en implémentation sur la *plate-forme Arduino UNO*. Du point de vue de la plate-forme primitive, elle est perçue comme une surcouche applicative. La plate-forme incrémentée propose des services pour accéder à la détection de présence et au

3. Pour l'Arduino UNO, une broche est dans l'état BAS si la tension d'entrée de la broche est inférieure à 3 volts, et dans l'état HAUT si la tension d'entrée de la broche est supérieure à 3 volts.

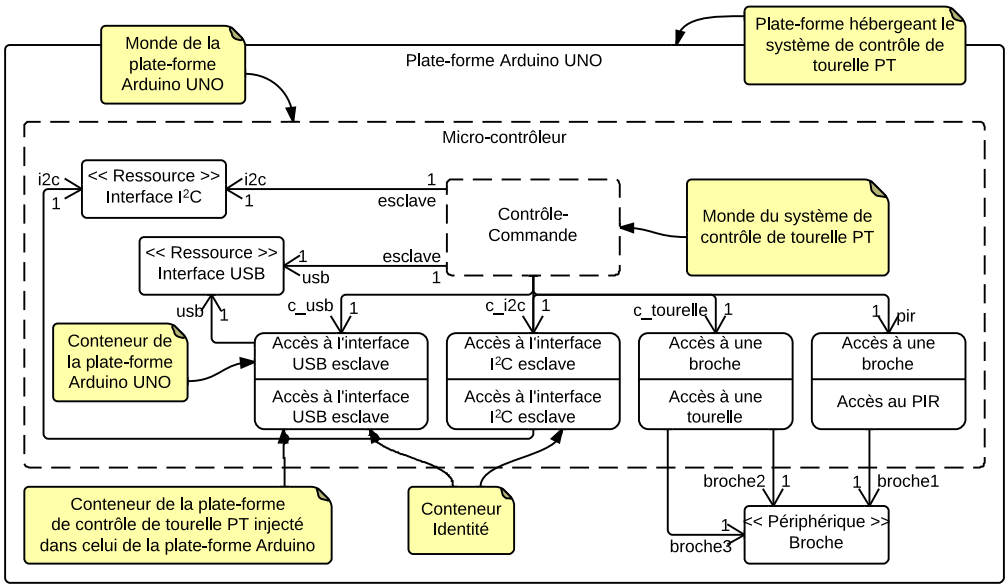


FIGURE 86 – Modèle d'implémentation du système de contrôle de tourelle PT

pilotage d'une tourelle, tandis que la plate-forme primitive ne permettait que la commutation de broches. Le modèle d'*implémentation de la plate-forme de contrôle de la tourelle PT* est illustré par la figure 86. Puisque la plate-forme Arduino UNO ne proposait qu'un seul monde, il n'y a pas de distribution possible, tous les objets du modèle sont inclus à l'intérieur du monde.

Dans ce modèle, nous avons injecté les conteneurs de la plate-forme de contrôle de tourelle PT dans les conteneurs de la plate-forme Arduino UNO. Ainsi, le conteneur d'accès au PIR est injecté dans un conteneur d'accès à une broche, tandis que le conteneur d'accès à la tourelle est injecté dans deux conteneurs d'accès pour une broche [4]. Chacune des deux broches sera utilisée pour le contrôle d'un des deux moteurs permettant l'orientation de la tourelle PT.

Afin de concevoir le *modèle d'implémentation de la plate-forme de contrôle de tourelle PT*, nous faisons également le choix de « *conserver* » les conteneurs de communication avec les interfaces I²C et USB en utilisant des conteneurs « *identité* » – en référence à la fonction mathématique *Identité*, c'est-à-dire des conteneurs n'affectant pas le comportement de l'objet injecté mais se contenant de l'injecter dans les conteneurs de la plate-forme primitive.

Nous définissons un conteneur « *identité* » comme un conteneur n'ayant pas de règle d'implémentation pour ré-écrire le comportement de l'objet. Dans le modèle d'analyse de la *plate-forme de contrôle de tourelle PT*, illustré par la figure 85 de la page 140, les conteneurs d'accès aux interfaces USB et I²C sont des conteneurs *identité*. Ils n'impactent pas le comportement des objets qui seront injectés à l'intérieur de ces derniers, mais se contentent de les injecter à leur tour dans les conteneurs d'accès aux interfaces USB et I²C la *plate-forme Arduino UNO*.

Plus formellement, en considérant un objet o (et son comportement fsm_o) injecté dans le conteneur *identité* c_{ID} (et sa règle d'implémentation *identité* t_{ID}) de la plate-forme p_1 et en considérant que ce dernier est lui-même injecté dans le conteneur c_2 (et sa règle d'implémenta-

4. L'injection dans deux conteneurs de la *plate-forme Arduino UNO* sera le seul cas où nous nous autoriserons à injecter un objet dans plusieurs conteneurs de la plate-forme.

tion t_2) de la plate-forme p_2, alors, par transitivité (cf. eq. 7.1), l'équation 7.2 définit l'injection de l'objet o dans le conteneur c_2 de la plate-forme p_2.

$$\forall \langle o,\, fsm_o \rangle, c_{ID}, c_2, \begin{cases} o \mathcal{R} c_{ID} \wedge c_{ID} \mathcal{R} c_2 \implies o \mathcal{R} c_2 \\ fsm_o \overset{t_{ID}}{\longmapsto} fsm_o \overset{t_2}{\longmapsto} \dfrac{\partial fsm_o}{\partial t_2} \end{cases} \tag{7.2}$$

Où $\dfrac{\partial fsm_o}{\partial t_2}$ est le comportement implémenté de o, ré-écrit par la règle d'implémentation t_2 du conteneur c_2 sur la plate-forme p_2. Ainsi, le conteneur *identité* d'une plate-forme incrémentée est un conteneur spécial permettant de « *remonter* » les conteneurs de la plate-forme primitive.

Enfin, en ce qui concerne les onze broches non utilisées [5], nous faisons le choix de *brider* leurs utilisations sur le système qui sera bâti sur la plate-forme incrémentée. Ceci s'effectue simplement en ne modélisant aucun conteneur *identité* (comme pour les périphériques de communication) pour permettre leur utilisation sur la *plate-forme de contrôle de tourelle PT*.

La *plate-forme de contrôle de tourelle PT* ainsi modélisée offre donc des services pour la détection de présence par l'intermédiaire d'un capteur de présence PIR et le contrôle d'une tourelle PT. En outre, elle permet également d'accéder aux ressources de communication USB et I^2C. La tourelle PT modélisée laisse entendre que la plate-forme doit être composée avec une autre plate-forme offrant des services de détection, par l'intermédiaire d'un capteur ou un actionneur, tel qu'une caméra ou un sonar, qui serait monté sur la tourelle.

L'exemple présenté permet de conclure sur les bénéfices apportés par la composition *par incrément*. Elle permet de construire des plates-formes évoluées à partir de plates-formes primitives. La plate-forme Arduino UNO aurait pu servir pour bâtir une autre plate-forme dédiée à un autre domaine d'activité que la surveillance. Cet aspect montre la *réutilisabilité* possible d'un niveau de plate-forme dans différents projets. Le fait de concevoir un incrément sur une autre plate-forme permet non seulement de l'adapter, mais aussi de la restreindre, comme nous l'avons fait avec les broches inutilisées de la *plate-forme Arduino UNO*. L'incrément présente des avantages en terme d'ingénierie, par exemple dans les chaînes de production permettant de concevoir une gamme d'un produit basé sur la même plate-forme dont certaines fonctionnalités seraient bridées sur les modèles les plus économiques.

En résumé

- ✓ *La composition par incrément permet de concevoir une plate-forme dite incrémentée sur la base d'une autre dite primitive*
- ✓ *Elle permet de définir de nouveaux services, d'adapter ou d'empêcher l'accès aux services de la plate-forme dite primitive*
- ✓ *Elle favorise la réutilisabilité, accentue le parallélisme des développements et favorise l'organisation efficace des équipes*
- ✓ *Elle permet un raffinement automatique de l'application selon les différentes plates-formes*

5. Nous avons injecté les conteneurs d'accès au PIR ainsi qu'à la tourelle dans trois des quatorze conteneurs d'accès à une broche de la *plate-forme Arduino UNO*. Il reste donc onze des quatorze broches non encore utilisées ainsi que les onze conteneurs associés.

2.3 Composition par assemblage

La composition *par assemblage* consiste à former une plate-forme en assemblant deux plates-formes (ou plus) mises les unes à côté des autres. Par commodité, nous appellerons *plate-forme assemblée* la plate-forme issue de l'assemblage de plusieurs plates-formes et *plates-formes composantes* les plates-formes qui sont composées. Conceptuellement, la plate-forme assemblée permet d'assurer l'interfaçage entre les différents périphériques des composantes, dans le cas où les plates-formes partagent des interfaces compatibles. Afin que les interfaces soient compatibles, il est nécessaire que les deux périphériques puissent être connectés par une ou plusieurs associations afin de pouvoir assurer la communication par envoi et réception de messages.

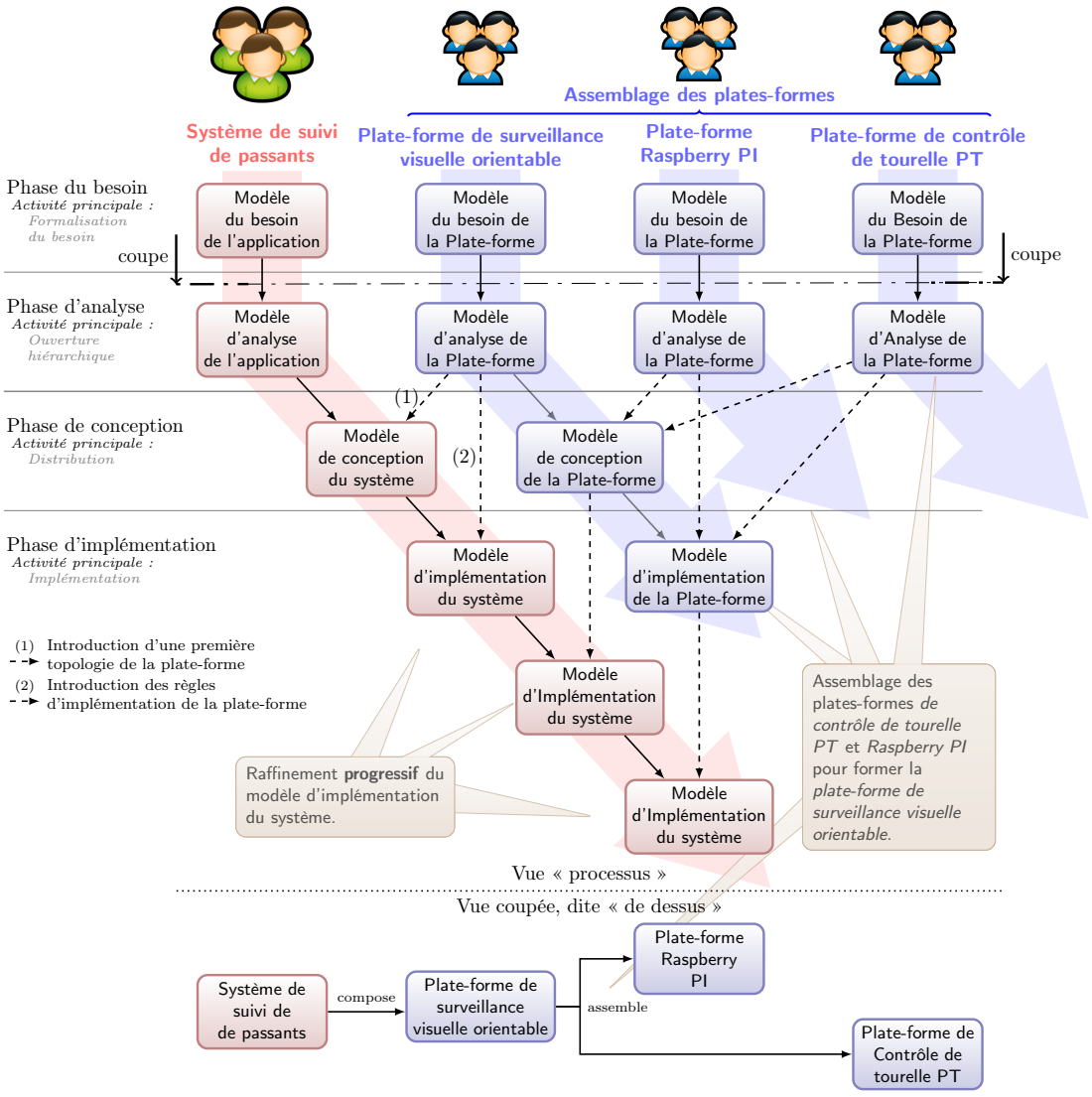


FIGURE 87 – Composition par assemblage

La composition par assemblage au sein du processus. La figure 87 illustre la composition *par assemblage* d'une plate-forme à partir de la définition de deux autres. Le flot de développement de la plate-forme assemblée est illustré à gauche tandis que les flots de développement des plates-formes composées sont situés à droite. Le flot de développement du système est illustré en rouge tout à gauche de la figure. Dans cet exemple, la plate-forme assemblée est la *plate-forme de surveillance visuelle orientable*. Elle compose *par assemblage* les *plates-formes de contrôle de tourelle PT* et *Raspberry PI*. La *plate-forme de surveillance visuelle orientable* est la plate-forme de plus haut niveau sur laquelle sont hébergés les objets applicatifs du *système de suivi de passants*.

Ce processus illustrant la composition *par assemblage* reprend les mêmes propriétés que le processus illustrant la composition *par incrément* (cf. fig. 80). Il offre les même bénéfices que ce premier en termes de réutilisabilité, collaboration et parallélisation de développement. Il permet également de raffiner le modèle de la plate-forme assemblée en phase de *conception* et d'*implémentation du système*, ce qui se traduit par un raffinement *automatique* du modèle d'implémentation du système hébergé sur la plate-forme assemblée. Les points de synchronisation au sein du processus permettent toujours de connaitre le moment à partir duquel les branches sont synchronisées. Par exemple, pour entamer le modèle de conception de la *plate-forme de surveillance visuelle orientable*, il est nécessaire d'avoir préalablement modélisé les modèles d'*analyse des plates-formes de contrôle de tourelle PT* et *Raspberry PI*. Tout comme pour la composition par incrément, la figure 87 illustre la vue « *de dessus* » par un plan de coupe de la vue « *processus* ». Dans cette figure, nous retrouvons le système de suivi de passants à gauche, ainsi que la composition par assemblage des plates-formes à droite.

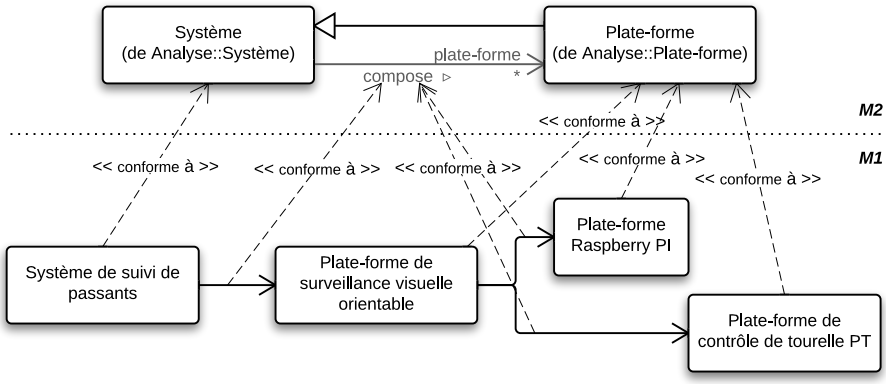


FIGURE 88 – Composition par assemblage au sein du langage

La composition par assemblage au sein du langage. Du point de vue du langage, la cardinalité « * » dans le *méta-modèle de composition* présenté en début de chapitre assure de pouvoir assembler deux plates-formes (ou plus) pour en former une autre. La figure 88 illustre un diagramme d'objets illustrant l'instanciation du processus pour respecter cette composition. Le système de suivi de passants *compose* la *plate-forme de surveillance visuelle orientable*, cette dernière assemblant les *plates-formes de contrôle de tourelle PT* et *Raspberry PI*.

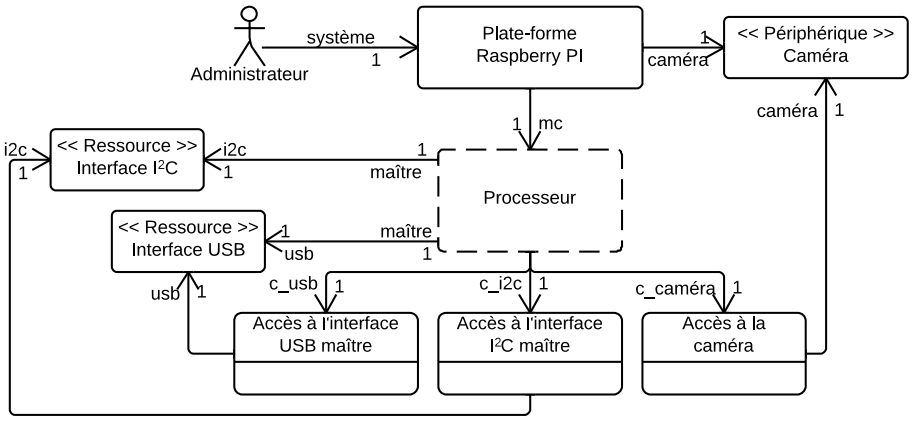


FIGURE 89 – Modèle d'analyse de la plate-forme Raspberry PI

Assemblage des plates-formes de contrôle de tourelle PT et Raspberry PI. Cette partie discute de l'assemblage des *plates-formes de contrôle de tourelle PT* et *Raspberry PI* pour former la *plate-forme de surveillance visuelle orientable*, telle qu'illustrée par la figure 79 de la page 134. La plate-forme assemblée permet le contrôle des différents périphériques offerts par les deux plates-formes mises côte à côte. Les différentes ressources de communication USB et I^2C sont assemblées de sorte que les deux plates-formes puissent communiquer entre elles.

Nous avions précédemment introduit le modèle d'*analyse de la plate-forme de contrôle de tourelle PT* (cf. fig. 85 page 140). La figure 89 illustre le modèle *partiel*[6] d'*analyse de la plate-forme Raspberry PI*. Cette plate-forme se compose d'une *caméra*, et des interfaces I^2C et USB afin de pouvoir la connecter à d'autres périphériques. Elle offre également le monde Processeur, pour héberger des objets. Enfin, des conteneurs sont disponibles pour assurer la bonne utilisation de la caméra, ainsi que l'accès aux ressources de communication I^2C et USB.

Le modèle d'*analyse de la plate-forme de surveillance visuelle orientable* est illustré dans la figure 90. Nous avons choisi de modéliser deux mondes délimitant des domaines orthogonaux. Le premier monde est dédié au domaine de l'imagerie tandis que le second est dédié au contrôle de capteurs et d'actionneurs. Un périphérique est modélisé pour établir une communication avec le second monde. Il se nomme *média de communication*. Il est utilisé afin de permettre la communication entre les objets applicatifs hébergés sur un premier monde vers le second. De ce fait, des conteneurs d'accès au média de communication sont modélisés dans chacun des mondes. Une ressource Installateur lie les deux mondes entre eux. Dans l'ingénierie logicielle, un *installateur* désigne un programme permettant d'installer un logiciel. Cette association est spécifique au déploiement d'applications sur les mondes de la plate-forme. Lorsqu'un utilisateur décide d'installer une application, le premier monde Traitement d'image réceptionne l'application sous la forme d'une archive, l'extrait dans un dossier, et déploie la partie de l'application relative au contrôle / commande sur le second monde qui le réceptionne. Enfin, nous avons modélisé tous les périphériques et conteneurs nécessaires à la surveillance et au traitement d'image.

La ressource Installateur permet de satisfaire un cas d'utilisation propre à la plate-forme

6. La documentation complète du processeur graphique du Raspberry PI, le *VideoCore IV*, jusqu'alors non libre, ne fut libérée que tardivement le 28 février 2014. Par conséquent, nous ne l'avons utilisé ni dans nos modèles, ni dans nos développements.

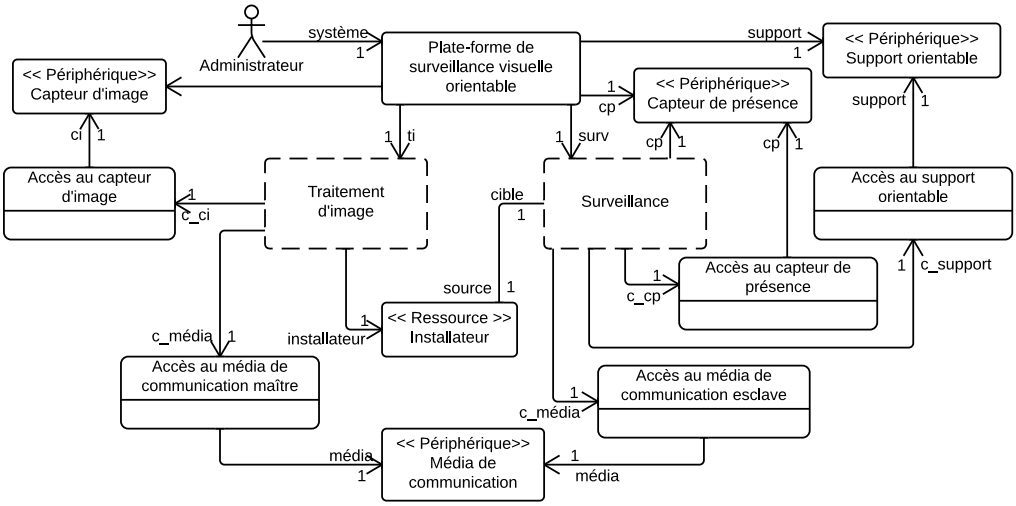


FIGURE 90 – Modèle d'analyse de la plate-forme de surveillance visuelle orientable

qui est l'installation d'applications. De ce fait, nous ne la modélisons pas comme un média de communication de la plate-forme et ne lui définissons pas de conteneurs d'accès puisque cette ressource n'est pas accessible par les objets applicatifs du *système de suivi de passants* (il ne s'agit donc pas d'un service de la plate-forme pour l'application qu'elle héberge). Il s'agit d'une ressource interne au monde, à la différence du média de communication. Comme tout autre objet, cette ressource peut-être distribuée dans les mondes des *plates-formes composantes* et injectée dans les conteneurs de ces dernières.

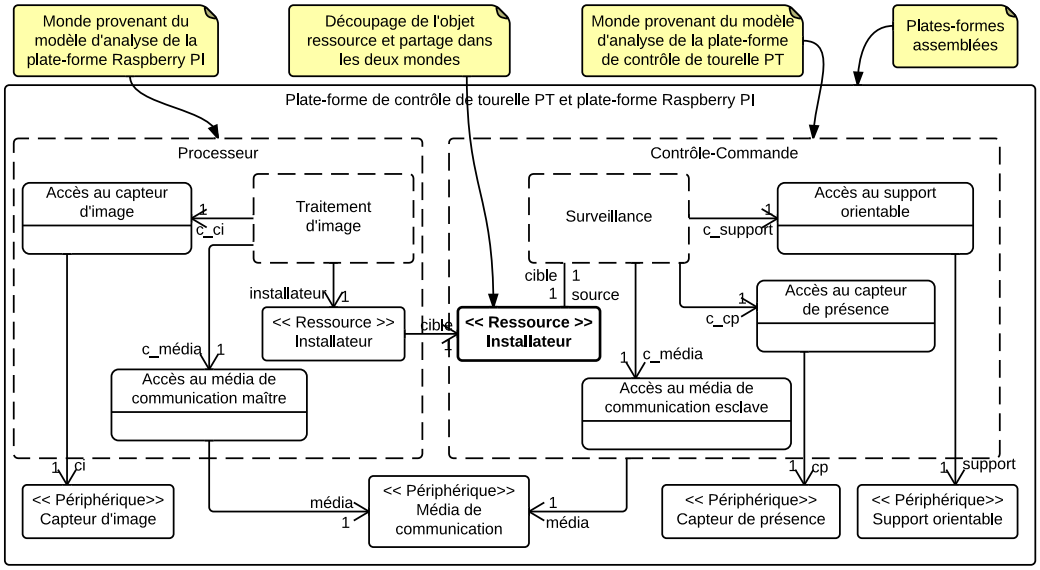


FIGURE 91 – Modèle de conception de la plate-forme de surveillance visuelle orientable sur les plates-formes de contrôle de tourelle PT et Raspberry PI

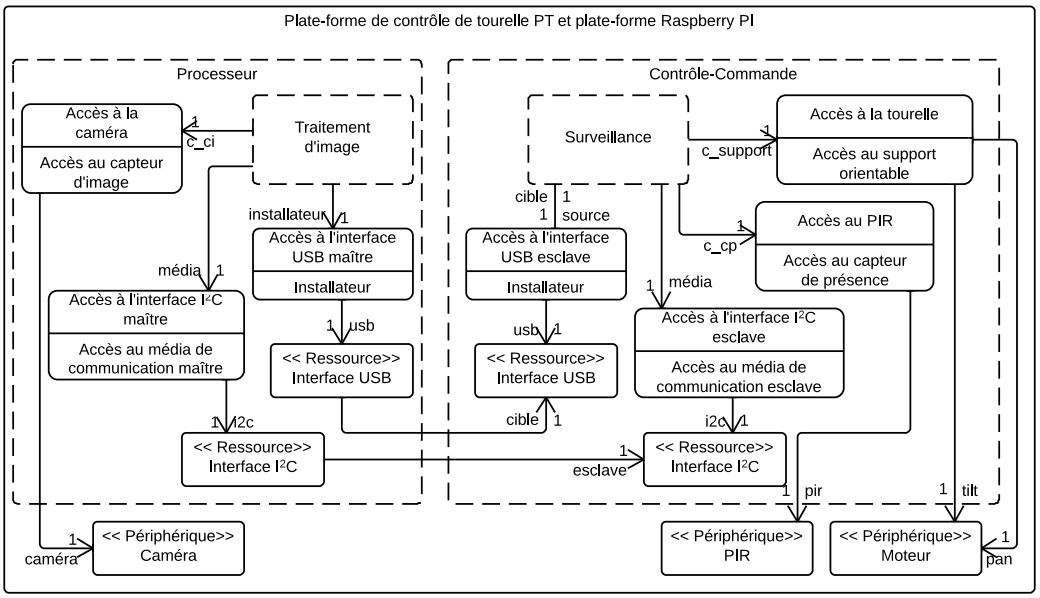


FIGURE 92 – Modèle d'implémentation de la plate-forme de surveillance visuelle orientable sur les plates-formes de contrôle de tourelle PT et Raspberry PI

Le *modèle de conception de la plate-forme de surveillance visuelle orientable* est illustré dans la figure 91. Dans ce modèle, nous avons hébergé les deux mondes respectivement dans les mondes des *plates-formes de contrôle de tourelle PT* et *Raspberry PI*. Les deux conteneurs d'accès au média de communication sont également hébergés sans distribution dans les mondes. En revanche, la ressource Installateur est découpée en deux et distribuée sur les deux mondes des deux *plates-formes composantes*. L'association modélisée entre les deux fragments de la ressource indique le sens de dialogue entre elles. La ressource hébergée dans le monde Processeur permet d'initialiser le dialogue avec la seconde qui ne peut que répondre.

Le modèle d'*implémentation de la plate-forme de surveillance visuelle orientable* est illustré dans la figure 92. Nous avons fait le choix d'injecter les conteneurs d'accès au média de communication dans les conteneurs d'accès aux interfaces I^2C. Les deux fragments de la ressource Installateur sont quant à eux injectés dans les conteneurs d'accès aux interfaces USB. Les conteneurs d'accès aux différents périphériques (conteneur d'accès au capteur de présence, conteneur d'accès au capteur d'image, conteneur d'accès au support orientable) sont injectés dans les conteneurs offerts par les deux plates-formes composantes (conteneur d'accès au PIR, conteneur d'accès à la caméra, conteneur d'accès à la tourelle).

L'exemple présenté permet de conclure sur les bénéfices apportés par la composition *par assemblage*. Elle permet de construire des plates-formes évoluées à partir de l'assemblage de *plates-formes composantes*. L'assemblage nécessite que les périphériques permettant d'assembler les deux plates-formes soient compatibles. Cette compatibilité nécessite l'interfaçage des périphériques de la plate-forme. Par exemple, le comportement de l'interface I^2C illustré dans le chapitre par la figure 58b de la page 109 illustrait une interface compatible. Configurée en mode *maître*, cette interface peut communiquer avec un ensemble d'interfaces configurées en mode *esclave*.

L'assemblage de plates-formes est possible avec l'extension que nous avons réalisée sur le méta-modèle de *composition*. Du point de vue du processus, les modèles d'analyse des différentes plates-formes composantes peuvent être réalisés par des équipes de développement différentes et le développement peut alors être effectué en parallèle. En outre, une plate-forme composante peut être utilisée dans différents assemblages permettant ainsi sa réutilisation. Pour les mêmes raisons que pour la composition par incrément, le raffinement du modèle d'implémentation d'un système hébergé sur une plate-forme composée *par assemblage* peut être réalisé de manière automatique, en considérant la transitivité de la relation d'injection dans les conteneurs.

En résumé

- ✓ *La composition par assemblage permet de concevoir une plate-forme dite assemblée à partir d'autres plates-formes dites composantes*
- ✓ *Elle permet de s'assurer que les plates-formes composantes peuvent être assemblées au moyen de périphériques compatibles*
- ✓ *Elle offre les mêmes avantages que l'incrément en termes de réutilisabilité, parallélisme et organisation*
- ✓ *Elle permet d'automatiser le raffinement de l'application selon les différentes plates-formes*

2.4 Enchainer les compositions

Nous avons présenté deux manières de composer les plates-formes dans cette partie, *incrément* et *assemblage*. Le méta-modèle *de composition* présenté au début de ce chapitre permet de combiner ces deux façons de composer afin de produire des plates-formes complexes. L'usage de ces deux compositions combinées est illustré dans la figure 79 de la page 134 pour implémenter le *système de suivi de passants*.

La composition mixte au sein du processus. Du point de vue du processus, la figure 93 illustre comment il est possible de combiner les deux types de composition. La branche la plus à gauche illustre le développement du *système de suivi de passants*, obtenu par assemblage des deux *plates-formes de contrôle de tourelle PT* et *Raspberry PI*. Le développement de la *plate-forme Raspberry PI* est illustré par la troisième branche du processus en partant de la gauche. La plate-forme est considérée comme finale, son développement se termine donc avec son modèle d'*analyse de la plate-forme*. Il aurait pu être envisagé de continuer le développement de cette plate-forme en la considérant comme l'incrément de la *plate-forme Raspberry PI nue*, qui ne possèderait pas de caméra et ne serait donc pas adapté au *traitement d'image*. Le développement de la *plate-forme de contrôle de tourelle PT* est illustré par la quatrième branche du processus. Cette plate-forme est composée *par incrément* de la *plate-forme Arduino UNO*, tout à droite de la figure 93. Cette dernière est considérée comme finale.

Ce processus illustre l'impact d'une composition sur plusieurs niveaux sur le raffinement du modèle d'*implémentation du système de suivi de passants*. Le nombre de modèles d'implémentation du système obtenus par raffinement automatique est lié au nombre de niveaux de

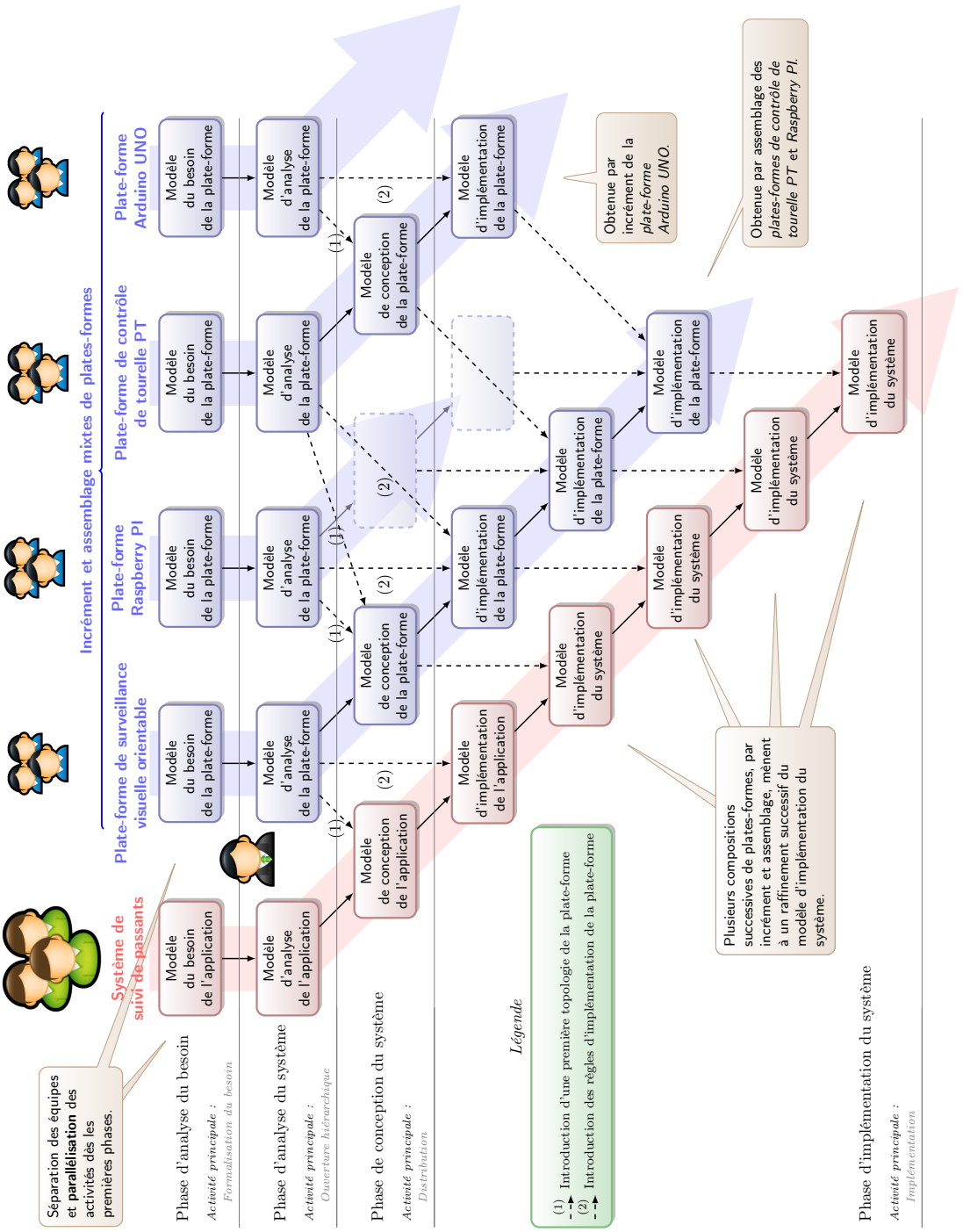


FIGURE 93 – Composition mixte par assemblage et incrément

plates-formes considérés. Ainsi, en considérant les *plates-formes de contrôle de tourelle PT* et *Raspberry PI* comme finales, le modèle d'implémentation du *système de suivi de passants* aurait subi deux raffinements automatiques. Puisque la *plate-forme de contrôle de tourelle PT* n'est pas finale, elle répercute son raffinement par effet « boule de neige » en phase d'*implémentation du système* sur les branches de gauche. Ainsi, le modèle d'*implémentation de la plate-forme de surveillance visuelle orientable* est automatiquement raffiné deux fois, celui du système de suivi de passants l'est quatre fois. L'illustration que nous avons choisie pour notre processus montre que chacune des branches se « décalent » vers la droite. Cette distinction le distingue clairement d'un cycle de vie en Y [Luiz Fernando Capretz 2005] par exemple qui fusionne les branches fonctionnelle et technique en une branche centrale. Dans notre illustration, les branches se décalent pour montrer que l'application se trouve toujours à la verticale des plates-formes sur lesquelles elle est hébergée, des plus abstraites vers les plus concrètes. En outre, le fait que les branches ne fusionnent pas montre l'indépendance du développement de la plate-forme vis-à-vis de l'application qu'elle héberge.

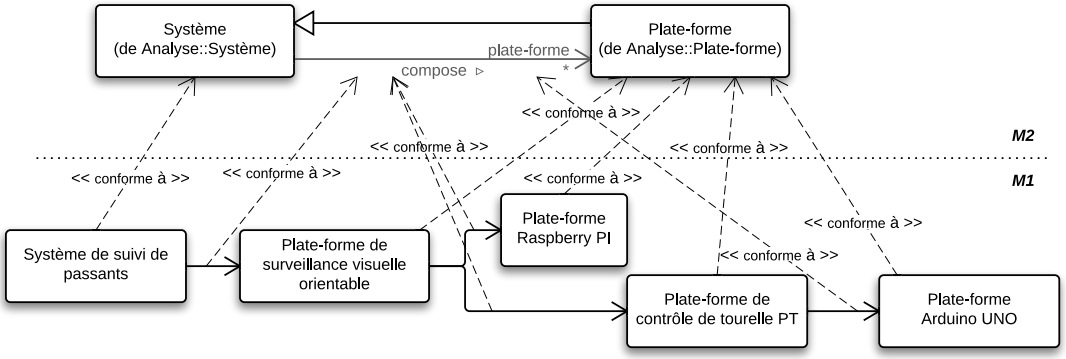


FIGURE 94 – Composition mixte au sein du langage

La composition mixte au sein du langage. La composition mixte est possible au sein du langage $\langle \mathsf{HOE} \rangle^2$ et du méta-modèle de composition. La cardinalité « * » de l'association liant les concepts de *système* et de *plate-forme*, ainsi que la spécialisation du concept de *système* par le concept de *plate-forme* assurent les deux types de composition ainsi que leur combinaison pour former des modèles de plates-formes composées. La figure 94 illustre cette composition pour le développement de la *plate-forme de surveillance visuelle orientable* hébergeant l'application du *système de suivi de passants*.

Le choix des noms des périphériques et des ressources a été fait pour illustrer l'abstraction des différents niveaux de plates-formes entre eux. Pour la *plate-forme de surveillance visuelle orientable*, les objets Capteur de présence, Capteur d'image, Support orientable, Média de communication sont des abstractions pouvant posséder plusieurs implémentations. Par exemple, le média de communication ne spécifie pas le type de communication (point à point, centralisé, distribué) mis en place. Nous faisons le choix de l'implémenter par un bus série synchrone bidirectionnel half-duplex I^2C. Le *support orientable* ne précise pas l'orientation du support ni son moyen de pilotage. Son implémentation a été réalisée au moyen d'une *tourelle PT* et de deux moteurs. De même, le capteur de présence ne précise pas son implémentation (binaire, analo-

gique, par effet hall, etc.). Nous faisons le choix de l'implémenter par un capteur binaire PIR. Du point de vue des modèles, l'objet Capteur modélisé dans le modèle d'*analyse de la plate-forme de contrôle de tourelle PT*[7] est implémenté dans le modèle d'*analyse de la plate-forme Arduino UNO* par une broche numérique.

Cet aspect montre l'implémentation progressive des différents éléments composants les modèles des plates-formes lors de la composition. Chaque niveau de plates-formes permet d'apporter des contraintes d'implémentation supplémentaires au modèle. L'usage de différents niveaux de plates-formes permet aux objets du système d'atteindre progressivement une implémentation finale sur une plate-forme sur laquelle nous souhaitons générer du code.

En résumé

- ✓ *Les compositions par assemblage et incrément peuvent être combinées pour produire des plates-formes complexes*
- ✓ *Chaque niveau de plates-formes capture un degré d'abstraction*
- ✓ *Les deux compositions sont prises en compte par le processus et le langage* $\langle HOE \rangle^2$

Synthèse du chapitre

Dans ce chapitre, nous avons présenté deux façons de composer les plates-formes afin d'en développer des plus complexes. Cet aspect répond au besoin identifié dans le chapitre 3 relatif aux faiblesses des processus de développement pour la construction de plates-formes complexes. Le processus proposé permet de composer récursivement des plates-formes selon deux types de composition, l'*assemblage* et l'*incrément*. L'assemblage permet de composer plusieurs plates-formes et de les interfacer au moyen de leurs périphériques. L'incrément permet de construire une plate-forme offrant des services plus concrets à partir d'une plate-forme plus primitive.

La composition de plates-formes se distingue de l'assemblage de composants dans le sens où il est nécessaire d'expliciter comment une application est hébergée sur les différentes plates-formes et l'impact sur leur exécution. Cet aspect est couvert par les deux concepts de *monde* et de *conteneur*. Les mondes permettent l'hébergement d'objets dans des espaces délimités dans lesquels les objets partagent une sémantique commune. Dans le cadre de l'assemblage, les conteneurs sont définis pour spécifier comment les mondes servent d'interfaces pour les communications des objets qu'ils hébergent vers des objets d'autres mondes.

Un tel processus est complet puisqu'il définit précisément l'ensemble des points de synchronisation. Pour tirer tous les bénéfices de ce processus, il est nécessaire de fournir tous les éléments nécessaires au chef de projet pour l'orchestrer efficacement. Pour cela, nous introduisons dans le prochain chapitre la gestion de projet intégrée au langage $\langle \mathsf{HOE} \rangle^2$.

7. À distinguer du véritable capteur physique attaché à une broche de la plate-forme Arduino UNO.

Chapitre 8

Gestion de projet et traçabilité couplées dans le processus

Publications pertinentes à ce chapitre :

[**Hili *et al.* 2014a**] Nicolas Hili, Christian Fabre, Sophie Dupuy-Chessa et Dominique Rieu. *A Model-Driven Approach for Embedded System Prototyping and Design.* In IEEE International Symposium on Rapid System Prototyping (RSP'14) (part of ESWEEK'14), New Dehli, India, Octobre 2014

[**Hili *et al.* 2012a**] Nicolas Hili, Christian Fabre, Sophie Dupuy-Chessa et Stéphane Malfoy. *Efficient Embedded System Development: A Workbench for an Integrated Methodology.* In ERTS2 2012, Toulouse, France, Février 2012

Liste des sections

Dans le chapitre 4, nous avions évoqué comme faiblesses des méthodes de développement actuelles le manque de traçabilité et la gestion de projet découplée des modèles produits. Ce découplage ne permet pas de mesurer et de piloter efficacement l'avancement des développements. Sans réel couplage entre les objets produits et les activités ayant permis de les obtenir, l'usage d'outils externes pour surveiller et piloter l'avancement de développement devient inévitable [Bendraou *et al.* 2007]. L'usage de tels outils autonomes produit une gestion de projet plus difficile à exploiter car totalement découplée des artefacts développés. Nous abordons ces problèmes de traçabilité et de gestion de projet dans ce chapitre. Pour établir une gestion de projet intégrée efficace et couplée aux modèles de produits, le langage de modélisation du processus doit répondre à différents besoins : il doit permettre la modélisation de tous les éléments essentiels pour la construction d'un processus basé sur des produits et leurs états et des activités avec des pré et post-conditions sur les produits ; les acteurs du processus doivent pouvoir être modélisés ; les modèles doivent être exécutables et permettre d'historiser les tâches réalisées. Dans notre cas, nous avions privilégié dès le chapitre 6 le modèle et les diagrammes d'activités que nous avons étendus afin que les tâches puissent être instanciées. Cette instanciation

permet d'obtenir une historisation des tâches exécutées, permet de mieux coupler les activités aux produits et offre au chef de projet un moyen de mesurer et de piloter l'avancement du développement. Cet aspect sera décrit dans la suite.

Ce chapitre est structuré de la manière suivante : la section 1 introduit brièvement l'état de l'art de la gestion de projet ; la section 2 présente le langage de gestion de projet intégrée dans $\langle \mathsf{HOE} \rangle^2$; la section 3 exhibe la traçabilité obtenue et les bénéfices pour établir une gestion de projet fortement connectée ; la section 4 aborde la gestion de projet intégrée à la méthode.

1 Introduction à la gestion de projet

Avant d'introduire la gestion de projet et de présenter comment elle peut venir en aide aux processus de développement de systèmes embarqués, il est nécessaire de définir précisément ce qui se cache derrière la notion de projet. L'une des références les plus connues en matière de définition de gestion de projet et des bonnes pratiques associées est le guide PMBOK [Duncan 1996, Guide 2001, Schwalbe 2013], défini par le Project Management Institute (PMI) [Project Management Institute 2014]. Ce dernier est un institut proposant des standards et des certifications relatifs à la gestion de projet. Le guide PMBOK contient un ensemble très complet de notions permettant de définir les diverses dimensions de la gestion de projet.

Projet. Un projet est défini comme étant « *un effort temporaire entrepris pour créer un produit ou un service unique* » [Guide 2001]. Il est défini par un ensemble de caractéristiques permettant de définir la nature très spécifique d'un projet particulier. Le PMI définit trois caractéristiques majeures, qui sont la durée d'un projet, la taille des équipes et le nombre d'organisations impliquées dans le projet (cf. fig. 95). La durée des projets est définie dans le guide comme allant de quelques semaines à plusieurs années. La taille des équipes peut varier de quelques personnes à quelques milliers. Enfin, le projet peut être mono ou multi-organisationnel. Ces caractéristiques principales fournissent un espace très large dans lequel les projets peuvent être situés. La grande diversité des projets implique que la gestion de projet doit pouvoir s'adapter à tous les types et toutes les tailles de projet.

Gestion de projet. La gestion de projet est définie comme étant « *l'application des connaissances, compétences, outils et techniques aux activités du projet pour répondre aux spécificités d'un projet* » [Guide 2001]. PMBOK définit cinq grandes phases d'un projet : l'*initialisation*, la *planification*, l'*exécution*, le *pilotage* et la *fermeture*. La gestion de projet a pour objectif d'identifier et de répondre aux différents besoins, intérêts et attentes des différents parties prenantes du projet, du développeur au client final. Elle doit être réalisée en prenant en compte les différentes contraintes intervenant dans les différentes phases du projet. Le guide PMBOK [Guide 2001] définit une liste (non exchaustive) de six contraintes qui sont le *contour* (des objectifs), la *qualité*, la *planification*, le *budget*, les *ressources* et le *risque* [1].

Dictionnaire des concepts & méta-modèles de gestion de projet. Le guide PMBOK détaille l'ensemble des différents concepts permettant d'établir les bonnes pratiques de la gestion de projet. En tête, nous retrouvons des notions élémentaires telles que les rôles et les participants à un projet. Le projet est caractérisé par un ensemble de propriétés telles que son nom, ses

1. La première version du guide définissait en plus la *communication* et l'*intégration*, l'*approvisionnement*.

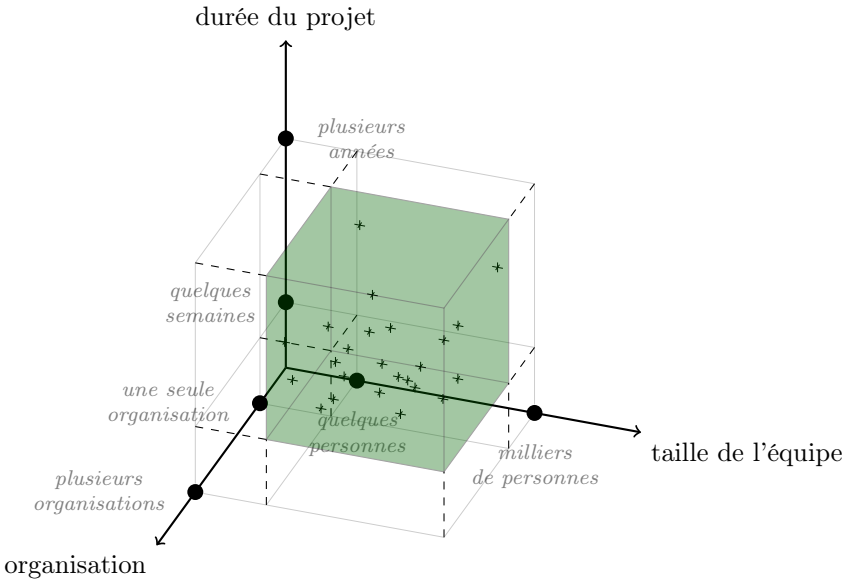


FIGURE 95 – Caractéristiques principales d'un projet [Guide 2001]

objectifs, le programme dans lequel il est intégré ou encore la ou les organisations en charge de ce programme. Les participants à un projet sont définis comme des ressources de ce dernier, au même titre que les ressources physiques. Les ressources physiques incluent l'ensemble des équipements nécessaires et remplissant les attentes des différents participants. Dans le cas des systèmes embarqués, il peut s'agir de salles ou de machines de calcul ou de simulation, dont le coût de fonctionnement peut occuper une place importante dans les coûts indirects du projet.

Le guide PMBOK est générique et non spécifique à l'industrie des systèmes embarqués ou à l'ingénierie du logiciel. Un second guide, nommé Software Engineering Body of Knowledge (SWEBOK) [Abran *et al.* 2001, Bourque *et al.* 2002] fut spécifiquement proposé pour l'ingénierie du logiciel. Dans ce guide, des notions spécifiques telles que les tests d'acceptation, les contraintes architecturales de conception ou les prototypes logiciels réalisés sont proposées. Il n'existe à notre connaissance aucune adaptation pour l'ingénierie des systèmes embarqués.

Callegari et Bastos ont identifié dans [Callegari & Bastos 2007] une taxonomie des principaux concepts de gestion de projet détaillés dans le guide PMBOK et en ont défini un méta-modèle. Ce dernier est illustré dans la figure 96. Dans ce méta-modèle, nous retrouvons un ensemble de concepts essentiels pour établir la gestion de projet. Les concepts d'activité et de phase sont présents, ainsi que les rôles et participants chargés de les réaliser ou les assigner.

Modèle de processus et couplage avec la gestion de projet. En ingénierie des systèmes d'information, la plupart des modèles de processus sont proposés avec quelques concepts de gestion de projet intégrés. C'est le cas de RUP qui est proposé sous la forme d'un méta-modèle comportant les concepts de rôle, d'activité, et d'outil permettant la réalisation d'une activité. Le couplage d'un méta-modèle permettant la modélisation de processus et un méta-modèle de gestion de projet peut donc amener à une duplication de concepts se chevauchant. La mise en commun de ces concepts permet de coupler un modèle de processus orienté activités et

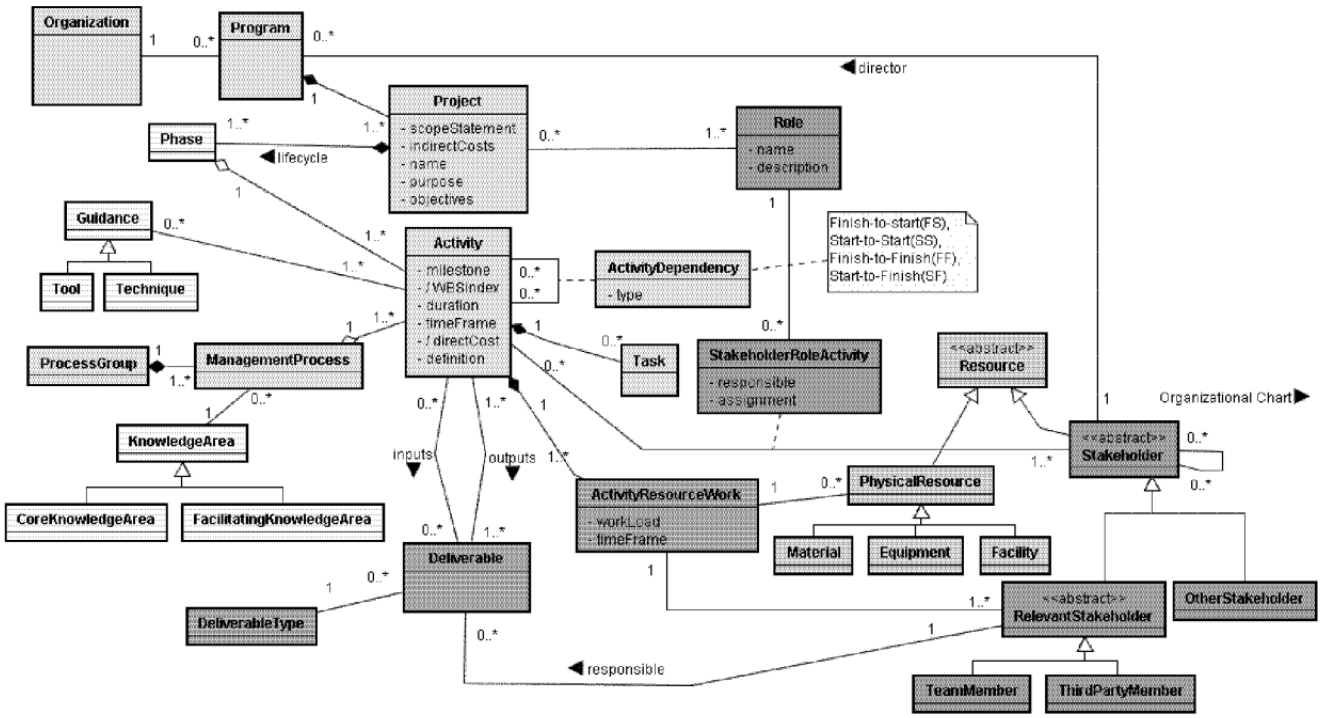


FIGURE 96 – Méta-modèle PMBOK [Callegari & Bastos 2007]

produits avec des concepts de gestion de projet. Plusieurs travaux se sont intéressés à établir un tel couplage avec des modèles de processus existants. Par exemple [Cottrell 2004, Callegari & Bastos 2007] proposent de coupler les concepts du guide PMBOK avec le processus RUP. Dans [Callegari & Bastos 2007], les auteurs présentent la réalisation d'un tel couplage. Il nécessite d'identifier les concepts communs aux méta-modèles de gestion de projet et des processus et de les regrouper dans un méta-modèle intermédiaire, permettant d'interfacer les méta-modèles en question. La faisabilité a été prouvée sur les méta-modèles de RUP et PMBOK et illustre la manière dont les processus peuvent être facilement couplés à une gestion de projet.

Positionnement de nos travaux. Les méta-modèles de gestion de projet présentés sont assez complets et suffisamment génériques pour être adaptés à plusieurs disciplines. Dans notre contribution, nous montrons comment un méta-modèle de gestion de projet peut facilement se coupler avec le processus $\langle \mathsf{HOE} \rangle^2$. Pour cela, nous avons repris les concepts essentiels de la gestion de projet afin de montrer les bénéfices tirés en termes de mesure et de pilotage de l'avancement du projet. Aussi, nous nous sommes principalement concentrés sur les aspects relatifs à la planification et à l'organisation des équipes et n'avons pas tenu compte des aspects relatifs aux coûts, ressources physiques, disponibilité des ressources physiques, risques et qualité. Les sections suivantes détaillent notre contribution.

2 Méta-modèle des activités et de la gestion de projet

Dans le chapitre 6, nous présentions le processus par phase sous la forme de diagrammes d'activités. Dans ces diagrammes, nous n'identifiions qu'un acteur que nous appelions *développeur*. Dans cette section, nous nous focalisons sur la formalisation du processus, en identifiant notamment les différents intervenants et nous formalisons les concepts de *phase*, de *tâche* et de *feuille de tâche* (i.e. instance d'une tâche particulière assignée à un développeur) en montrant comment ces dernières permettent une gestion de projet intégrée. La figure 97 présente le méta-modèle de la *gestion de projet*, que nous détaillons de haut en bas.

Phases et tâches. La partie haute du méta-modèle de la figure 97 permet de modéliser les *phases*, *tâches* et *feuilles de tâche* selon le processus ⟨HOE⟩[2]. Nous nous sommes appuyés sur la notation des diagrammes d'activités UML mais la sémantique appliquée est spécialisée. Le concept de *phase* correspond à l'une des quatre phases du processus. Elle se modélise à l'aide du concept d'*activité* UML et bénéficie donc de sa notation sous la forme d'un diagramme d'activités UML, telle que nous l'avons présentée pour modéliser les quatre phases du processus dans le chapitre 6. Une *phase* est composée d'un ensemble de *tâches*. Une tâche correspond à une activité du processus décrit dans le chapitre 6. Elle se modélise à l'aide du concept d'*action*

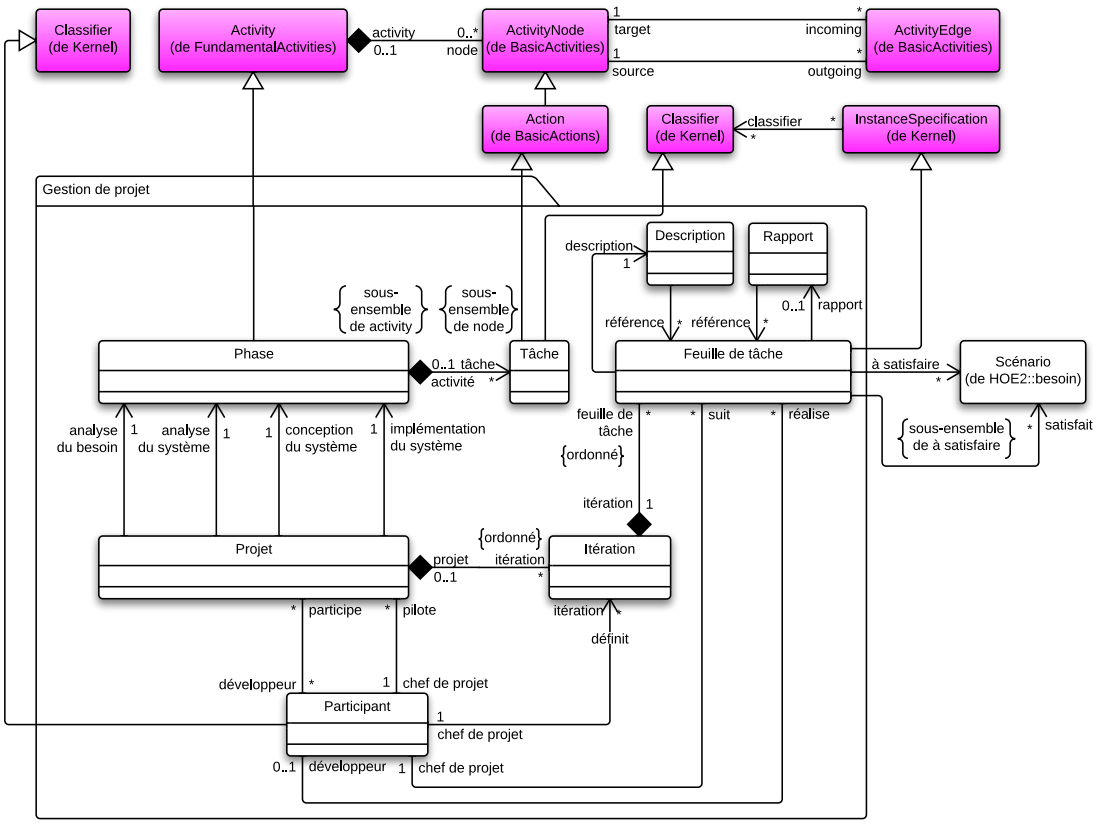


FIGURE 97 – Méta-modèle de gestion de projet

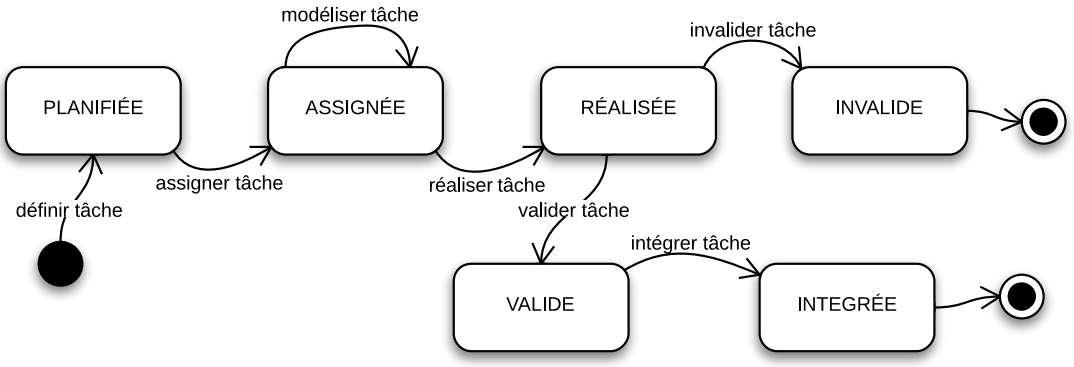


FIGURE 98 – Les états de la feuille de tâches

UML. Elle est définie par le *chef du projet* qui peut directement la réaliser (par exemple, l'initialisation ou la clôture des différentes phases et des modèles correspondants) ou bien l'assigner à un développeur (les tâches de modélisation durant les différentes phases). Une tâche étend la méta-classe UML *Classifier*. Une tâche peut alors être instanciée afin d'obtenir une *trace d'exécution*. Nous appelons ces instances les *feuilles de tâches*. Les *feuilles de tâches* sont définies par le *chef de projet*, assignées à un *développeur* et réalisées par ce dernier. La figure 98 illustre la machine à états de la feuille de tâches. Initialement, elle est dans l'état PLANIFIÉE, c'est-à-dire qu'elle est définie par le *chef de projet*. Elle référence un ensemble de scénarios à satisfaire et possède une *description*. Elle n'est pas encore assignée à un développeur. Lorsque le chef de projet l'assigne à un développeur, elle passe dans l'état ASSIGNÉE. Le développeur peut alors commencer à compléter la feuille de tâche. Une feuille de tâche est considérée RÉALISÉE lorsque tous les scénarios à satisfaire sont satisfaits. Le développeur peut alors la retourner, complétée avec un *rapport* de réalisation pour expliquer ou justifier son développement au chef de projet qui peut choisir de l'invalider (état INVALIDE) ou au contraire de la valider (état VALIDE). Enfin, dans le cas où la *feuille de tâche* a été préalablement validée, le chef de projet peut l'intégrer dans son modèle global. À chaque état correspond une date (création, assignation, réalisation, validation et intégration) qui permet d'évaluer l'évolution de la feuille de tâche.

La figure 99 illustre l'échange entre le chef de projet et le développeur. En premier lieu, le chef de projet *planifie une tâche* et l'*assigne* à un développeur. La feuille de tâche ASSIGNÉE est transmise au développeur qui la reçoit. Ce dernier doit alors la *réaliser*. Entre temps, le chef de projet est en mesure de planifier de nouvelles tâches à assigner à d'autres développeurs. Lorsqu'une feuille de tâche est RÉALISÉE par un développeur, ce dernier la retourne au chef de projet qui peut la *valider* ou l'*invalider*. Dans le cas d'une feuille de tâche VALIDÉE, le chef de projet peut l'intégrer dans le développement courant. La feuille de tâche est alors INTÉGRÉE.

Projet et itérations. Un *projet* permet la modélisation d'un système ou d'une plate-forme selon le processus $\langle \mathsf{HOE} \rangle^2$. Il est piloté par un *chef de projet* et réalisé par un ensemble de *développeurs*. Il offre une vision globale au chef de projet de ce qui a été réalisé et ce qu'il reste à faire. Il référence quatre activités correspondant aux quatre phases du processus. Le concept de *projet* permet d'organiser les différentes tâches dans un cycle d'*itération*. Le concept *Itération* permet de définir ce cycle dans lequel les différentes tâches à réaliser sont définies. Une

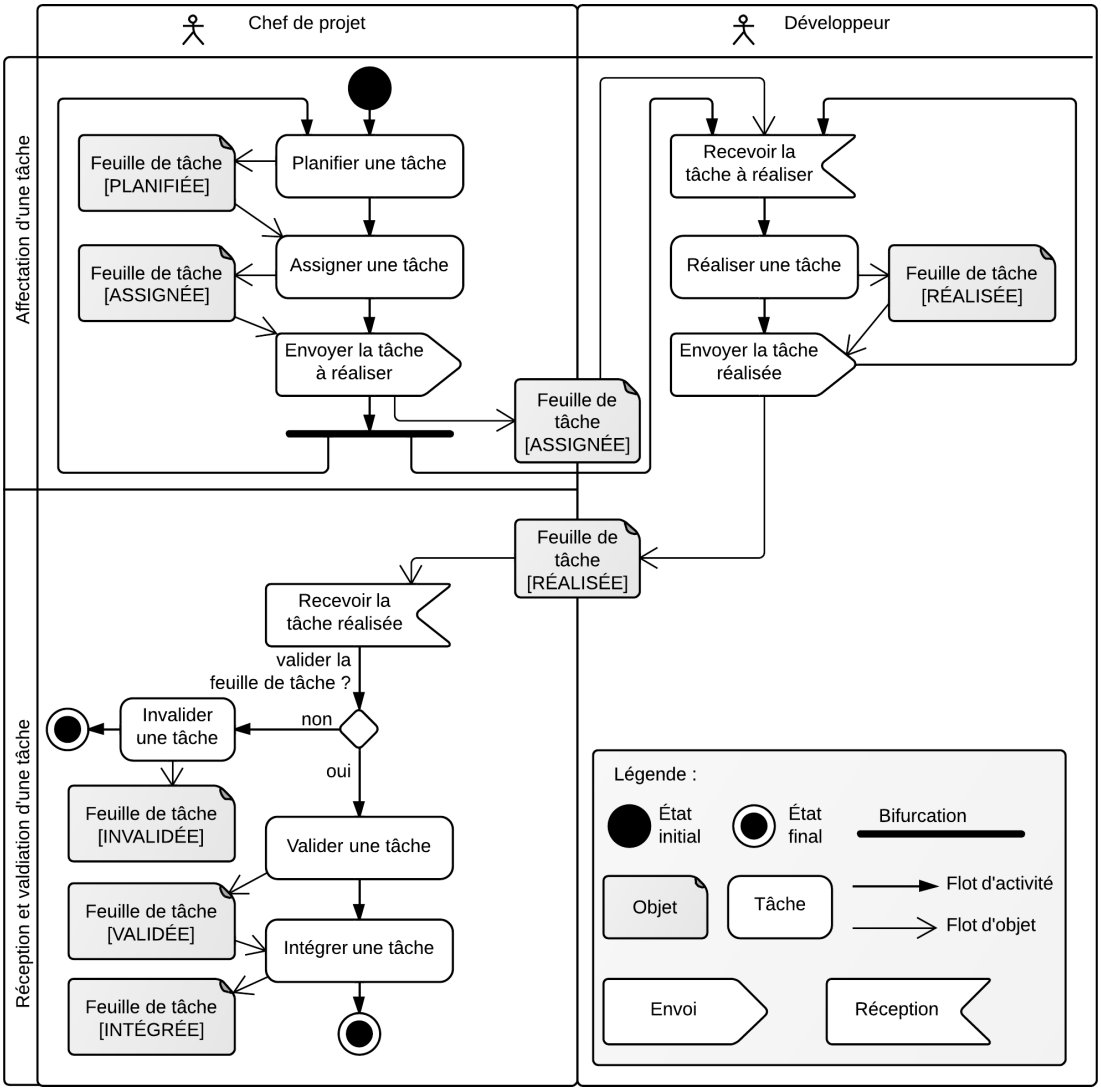


FIGURE 99 – Diagramme d'activités de la gestion d'une feuille de tâche

itération est définie par un chef de projet. Puisque les feuilles de tâches sont directement liées aux scénarios à satisfaire issus de l'analyse du besoin, le concept *Itération* permet au chef de projet de définir des itérations en sélectionnant des besoins prioritaires à satisfaire en premier. Cette façon de définir des itérations dans $\langle \mathsf{HOE} \rangle^2$ est inspirée de l'approche dirigée par les cas d'utilisation de la méthode USDP où à chaque démarrage d'une itération, les cas d'utilisation prioritaires sont sélectionnés et couverts en premier, afin de réduire les risques de développement. Cette approche est également utilisée dans des méthodes comme RAD permettant de réaliser rapidement des prototypes.

Id	Description
[1]	Un développeur ne peut réaliser qu'une tâche d'un projet auquel il participe.

```
context Participant inv:
  self.réalise.itération.projet->forAll(p : Projet | p.développeur->
    exists (p : Participant | p = self))
```

[2]	Un chef de projet ne peut suivre une tâche que d'un projet qu'il pilote.

```
context Participant inv:
  self.suit.itération.projet->forAll(p : Projet | p.chef de projet =
    self)
```

[3]	Un chef de projet ne peut définir une itération que d'un projet qu'il pilote.

```
context Participant inv:
  self.définit.projet->forAll(p : Projet | p.chef de projet = self)
```

[4]	Un participant ne peut être à la fois développeur et chef de projet sur un même projet.

```
context Projet inv:
  self.développeur->excludes (p : Participant | p = self.chef de projet)
```

[5]	Le *Classifier* dont la feuille de tâche est une instance est de type *Tâche*.

```
context Feuille de tâche inv:
  self.classifier.oclIsTypeOf(Tâche)
```

[6]	Une feuille de tâche est réalisée quand tous les scénarios à satisfaire sont satisfaits.

```
context Feuille de tâche inv:
  self.oclIsInState('Réalisée') implies
  self.satisfait->includesAll(self.à satisfaire)
```

TABLE 31 – Liste des contraintes du méta-modèle de la gestion de projet

Participants. Nous proposons le concept de *participant* et identifions deux *rôles*. Un participant à un projet peut être soit un *chef de projet*, soit un *développeur*. Les *développeurs* peuvent être affectés à plusieurs *projets* et peuvent à ce titre réaliser des *tâches* de ce projet qui leur sont assignées par le *chef de projet*. Le *chef de projet* mesure et pilote quant à lui l'avancement du projet, définit les *itérations*, assigne les tâches aux *développeurs* et suit leur réalisation. Un chef de projet peut être développeur sur un autre projet et inversement.

La table 31 résume les contraintes sur le méta-modèle. La première contrainte permet d'assurer qu'une tâche ne peut être assignée à un développeur que si ce dernier participe au projet. La seconde contrainte similaire à la première permet d'assurer qu'une tâche ne peut être suivie par un chef de projet que si ce dernier pilote bien le projet dans lequel cette tâche est définie. La troisième permet d'assurer que le chef de projet ne peut définir des itérations que dans le projet qu'il pilote. La cinquième assure que les deux relations entre le projet et les participants sont mutuellement exclusives, c'est-à-dire qu'un chef de projet ne peut être développeur du même projet et inversement. La cinquième assure le lien entre la feuille de tâche et la tâche, la feuille de tâche ne pouvant être instance que d'une tâche. Cette contrainte peut s'exprimer du fait qu'une feuille de tâche est un sous-type de *InstanceSpecification* qui représente un *Classifier* UML et qu'une tâche est un sous-type de *Classifier*. Enfin, la dernière requête permet de définir l'état RÉALISÉE de la feuille de tâche. Cette dernière est dans l'état RÉALISÉE lorsque tous les scénarios de l'association « à satisfaire » sont inclus dans l'association « satisfaits ».

En résumé

- ✓ *Extension des diagrammes d'activités du chapitre 6 pour définir les rôles de chef de projet et de développeur*
- ✓ *Extension du méta-modèle d'activités pour historiser les tâches et connecter la gestion de projet aux produits développés*

3 Traçabilité forte entre le produit et le processus

Cette section détaille la traçabilité proposée au sein du langage et du processus ⟨HOE⟩[2]. Nous nous sommes intéressés à établir deux types de traçabilité. La première est une traçabilité *bidirectionnelle* permettant d'assurer le lien *dans les deux sens* entre les concepts de tâche et de produit modélisé durant la tâche. La seconde traçabilité est la traçabilité des besoins permettant de tracer tous les éléments des modèles réalisés et toutes les tâches ayant permis de les obtenir au regard des besoins initialement formalisés sous forme de scénarios.

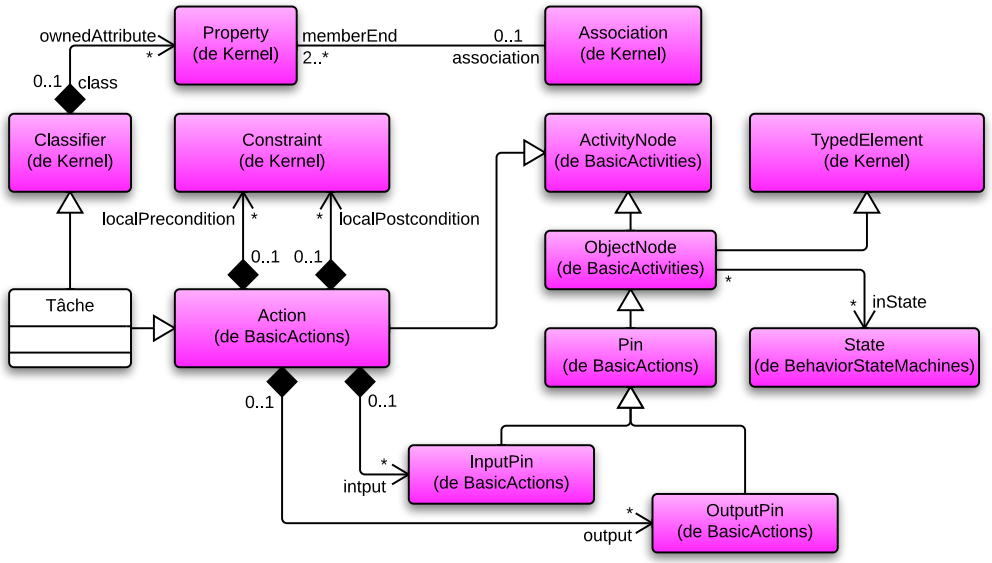


Figure 100 – Fragment du méta-modèle – tâche

Traçabilité de la tâche au produit. La première traçabilité permet à un chef de projet de découvrir l'ensemble des produits obtenus par la réalisation d'une feuille de tâche, instance d'une tâche particulière. Elle est directement induite du méta-modèle des activités et actions de UML. La figure 100 illustre une portion du méta-modèle UML. La méta-classe UML *Action* référence des objets entrants (*InputPin*) et sortants (*OutputPin*) de l'action. Ces méta-classes

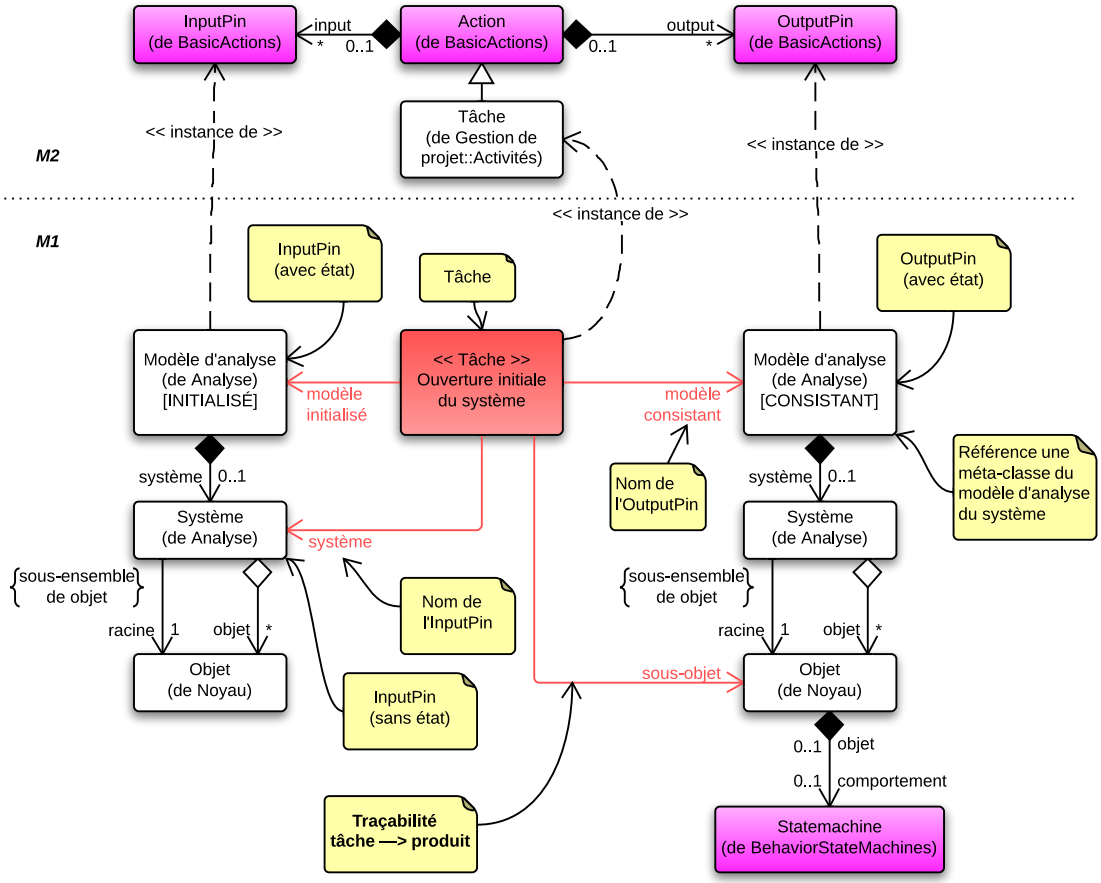


FIGURE 101 – Traçabilité de la tâche vers le produit

étendent la méta-classe *ObjectNode* permettant de référencer un objet dans un diagramme d'activités UML. Cet objet est typé (extension de la méta-classe *TypedElement*) et permet d'indiquer un ou plusieurs état(s) dans lequel l'objet référencé se trouve.

La figure 101 illustre une tâche du processus ⟨HOE⟩[2], permettant de réaliser l'ouverture initiale du système en phase d'analyse [2]. La traçabilité est illustrée par les flèches rouges issues de la tâche vers les objets entrants et sortants. Ces flèches permettent de définir le couplage des tâches vers les produits. Elles modélisent le lien entre les concepts *Action* et *Input/Output Pin*. La méta-classe *ObjectNode* étend la méta-classe *TypedElement* seulement et non *MultiplicityElement*. À ce titre, cette référence est *typée* mais non *dénombrable*. Le fait de proposer le concept de *tâche* étendant la méta-classe UML *Action* nous permet donc d'établir le lien au niveau M1 entre une tâche et le concept du méta-modèle produit (grâce au concept *OutputPin*) et le lien au niveau M0 entre la feuille de tâche et l'élément du modèle produit.

2. Nous avons choisi une légère variation de notation par rapport aux diagrammes d'activités UML afin d'en simplifier la lecture. Les objets entrants sont modélisés à gauche de la tâche, tandis que les objets sortants sont modélisés à droite de celle-ci.

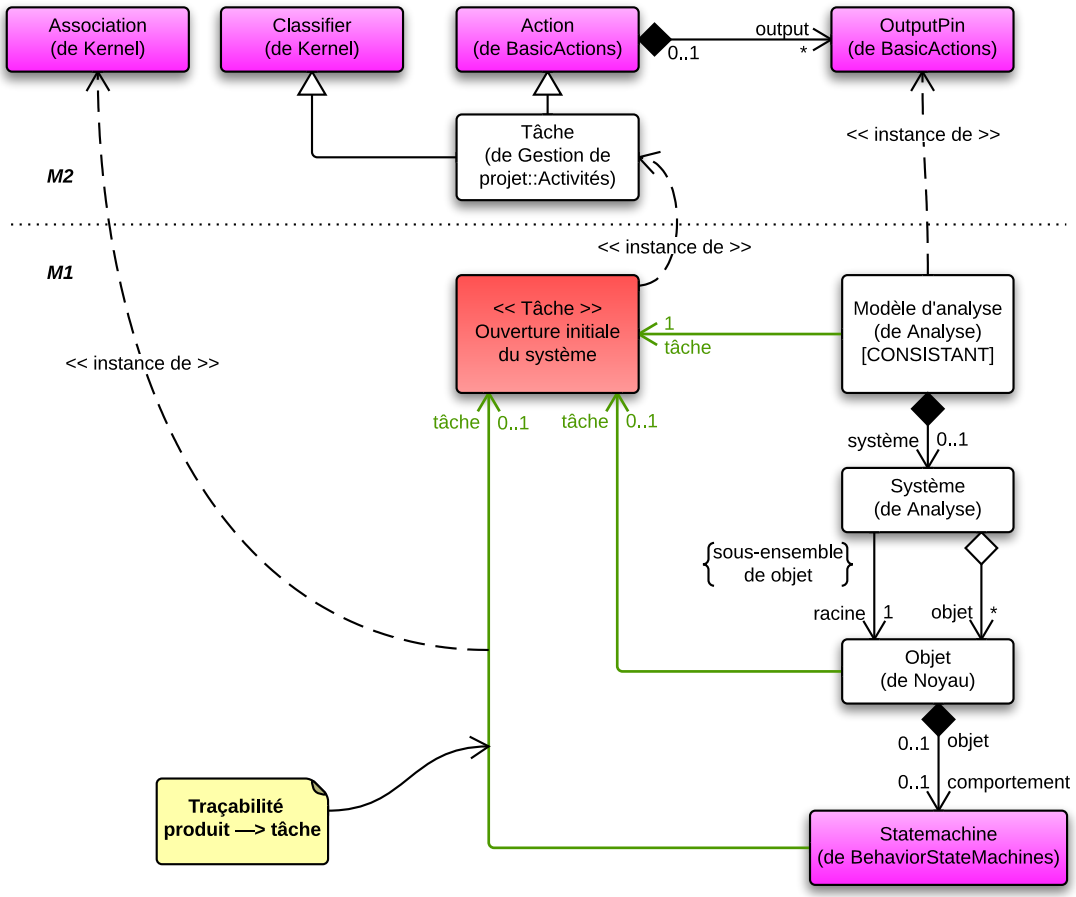


FIGURE 102 – Traçabilité du produit vers la tâche

Traçabilité du produit à la tâche. Il est désormais possible de découvrir l'ensemble des éléments produits associés à une feuille de tâche. La traçabilité inverse permet, à partir d'un élément du modèle, de découvrir l'ensemble des tâches qui ont permis de le modéliser. Cette traçabilité inverse est rendue possible grâce à notre extension. Le concept de *tâche* que nous proposons étend la méta-classe *Classifier*. À ce titre, en plus d'être instanciable, elle peut être associée à d'autres éléments par l'intermédiaire de la méta-classe UML *Association* (cf. fig. 100). Aussi, nous définissons pour chaque produit du langage $\langle \mathsf{HOE} \rangle^2$ une association du produit vers la ou les tâches permettant de l'obtenir.

Le fait de proposer le concept de *tâche* étendant la méta-classe UML *Classifier* nous permet donc de modéliser la traçabilité des produits vers les tâches. La figure 102 illustre la tâche d'ouverture initiale du système précédemment présentée dans la figure 101 en focalisant sur la traçabilité du produit vers les tâches. Cette traçabilité est illustrée par la flèche verte entre les objets sortants de la tâche et la tâche elle-même. La cardinalité permet de définir si le produit est obligatoirement obtenu à partir de cette tâche ou si d'autres tâches sont en mesure de le créer. Par exemple, le modèle d'*analyse du système* dans l'état CONSISTANT ne peut être obtenu

que par la tâche d'*ouverture initiale du système*. En revanche les objets du modèle d'*analyse du système* peuvent être obtenus à partir de cette tâche ou bien de la tâche d'*ouverture hiérarchique d'un objet*. Il est à noter que ces flèches sont des instances de la méta-classe UML *Association*. À ce titre, l'objet pointé par l'association possède un type, un nom de rôle ainsi qu'une cardinalité.

Cette première traçabilité bidirectionnelle présente des bénéfices importants pour le couplage entre le produit et une gestion de projet permettant au chef de projet de piloter et mesurer l'avancement du développement. Il peut avoir une vue d'ensemble du développement et sélectionner un élément du modèle pour connaitre les raisons (i.e. la feuille de tâche associée) de son développement. Il peut également parcourir l'arborescence de toutes les feuilles de tâches intégrées et en sélectionner une pour connaitre l'impact sur la vue d'ensemble du modèle (i.e. les objets modélisés durant cette tâche). Une telle traçabilité bi-directionnelle pourrait être très facilement intégrée dans un outil offrant une synchronisation directe et en temps-réel entre des vues de gestion de projet et des vues d'ensemble du modèle produit.

Traçabilité au besoin. Nous avons vu dans les deux parties précédentes que la traçabilité entre les produits et les tâches étaient bi-directionnelles et assurées par notre extension du langage d'activités UML. Dans cette partie, nous revenons sur le concept de *feuille de tâche* et montrons comment ce concept permet d'assurer la traçabilité au sein du processsus à l'aide des scénarios du modèle du *besoin*.

La figure 103 illustre la partie du méta-modèle relative à la feuille de tâche et leurs propriétés. Elle possède une *description* et un éventuel *rapport* de réalisation. Une *description* est rédigée par le *chef de projet* à destination du *développeur*. Un *rapport* est retourné par le *développeur* à qui la tâche a été assignée pour expliquer ou justifier son développement. Le rapport

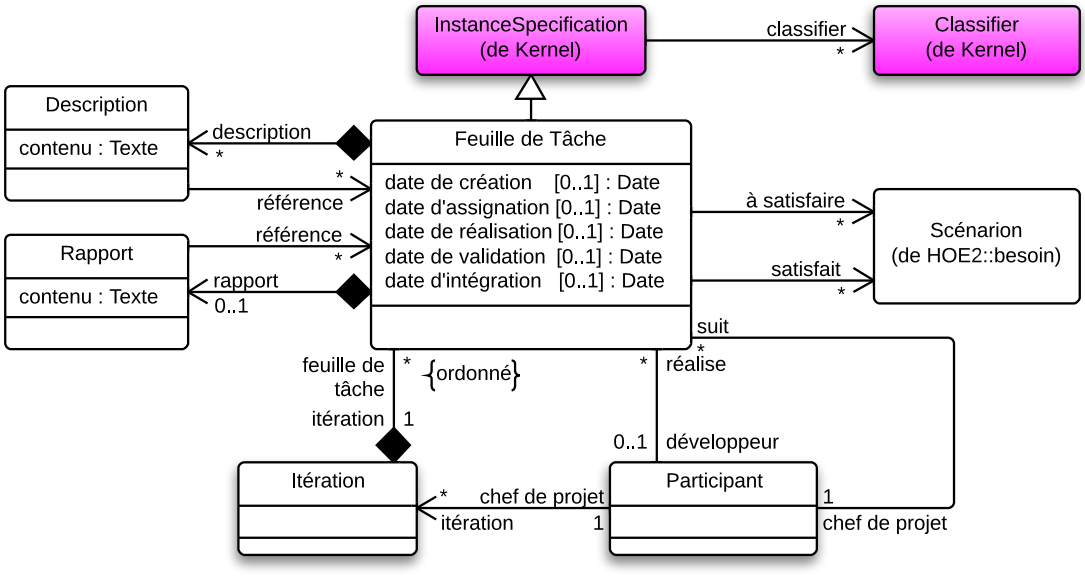


FIGURE 103 – Fragment du méta-modèle – feuille de tâche

et la description peuvent référencer d'autres *feuilles de tâches* (par exemple, la description d'une tâche de *complément d'ouverture hiérarchique* durant la phase d'*analyse du système* peut référencer la première tâche d'*ouverture hiérarchique* du même objet).

La traçabilité au besoin est assurée par les scénarios du besoin. La feuille de tâche référence un ensemble de *scénarios* du modèle *du besoin* à satisfaire et un sous-ensemble satisfait. Ainsi, il est possible à tout moment d'indiquer qu'une tâche doit satisfaire un ou plusieurs scénarios. Cela permet entre autres de définir le *niveau de satisfaction* d'un scénario (définir jusqu'à quelle phase le scénario est satisfait). Enfin, puisqu'une *feuille de tâche* est définie comme une instance d'une *tâche*, il est donc possible d'associer une feuille de tâche à un produit développé (selon la traçabilité bidirectionnelle entre les tâches et les produits définie plus tôt). *Ainsi, tout au long du processus et pour chaque objet modélisé, ou pour chaque tâche définie, il est possible de définir à quels besoins il renvoie.*

Les quatre tables 32, 33, 34 et 35 illustrent la modélisation de tâches pour l'analyse du besoin. Les tâches pour l'ensemble du processus seront détaillées en annexe C. Pour ces tâches, nous avons défini la personne en charge de la réaliser (le *développeur* ou le *chef de projet*), les pré-conditions opérant sur les produits en entrée et les post-conditions opérant sur les produits en sortie. Pour chaque tâche, un diagramme illustre la tâche (en rouge), la traçabilité de la tâche vers les produits (en rouge) et la traçabilité des produits vers la tâche (en vert). Lorsque le produit en entrée ou en sortie d'une tâche est un modèle, nous spécifions son état.

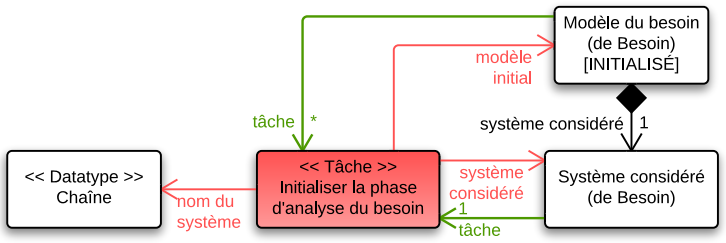
Tâche	**Description**
Initialiser la phase d'analyse du besoin	Initialise le modèle du besoin à partir du nom du système à concevoir. **Réalisation :** Chef de projet (automatisable). **Pré-condition :** un nom du système est donné **Post-condition :** modèle du besoin INITIALISÉ  **Explication :** Initialement, le modèle d'analyse du besoin n'existe pas. Il est alors créé en contenant un système considéré. Dans l'état INITIALISÉ, le modèle du besoin ne possède pas encore d'acteurs ni de cas d'utilisation. Il n'est pas suffisant pour initialiser le modèle d'analyse. Une fois l'initialisation effectuée, il peut désormais réaliser la tâche d'*élargissement de la formalisation du besoin.*

TABLE 32 – Gestion de projet : Tâche d'intialisation de la phase d'analyse du besoin

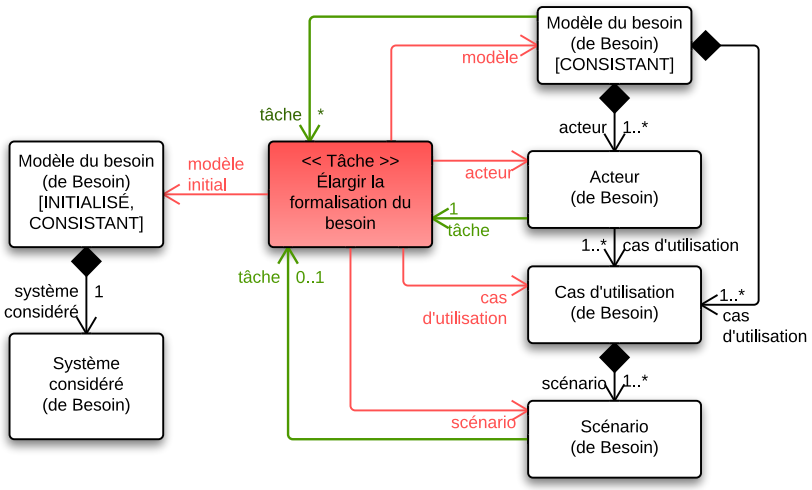
Tâche	Description
Élargir la formalisation du besoin	Élargit la formalisation du modèle du besoin en ajoutant des acteurs, cas d'utilisation et scénarios nominaux. Le modèle sortant est un modèle dans l'état CONSISTANT. **Réalisation :** Développeur. **Pré-condition :** modèle du besoin INITIALISÉ ou CONSISTANT **Post-condition :** modèle du besoin CONSISTANT  **Explication :** Initialement, le modèle d'analyse du besoin peut être dans l'état INITIALISÉ ou CONSISTANT. Cette tâche assure la création de nouveaux acteurs, cas d'utilisation et scénarios nominaux, afin que le modèle reste ou devienne CONSISTANT. Il peut produire un scénario. La cardinalité 0..1 pour l'association entre le *scénario* et la *tâche* se justifie par le fait que ce dernier peut également être produit par une tâche d'*extension de la description du besoin modélisé*. Le modèle étant dans l'état CONSISTANT, il est désormais possible d'initialiser le *modèle d'analyse du système*.

TABLE 33 – Gestion de projet : Tâche d'élargissement de la formalisation du besoin

Cette modélisation, commune à tous les modèles $\langle \mathsf{HOE} \rangle^2$ est illustrée par la figure 104. Dans $\langle \mathsf{HOE} \rangle^2$, nous avons identifié les états d'un modèle et trois types de tâches, permettant de faire varier l'état du modèle, illustré par la figure 104. Le modèle est d'abord INITIALISÉ par une tâche d'*initialisation*. Il devient CONSISTANT dès le moment où une première tâche de *modélisation* a lieu. Un modèle CONSISTANT est suffisant pour initialiser les phases suivantes. La tâche de *clôture* effectuée quand le modèle est considéré comme terminé le rend COMPLET.

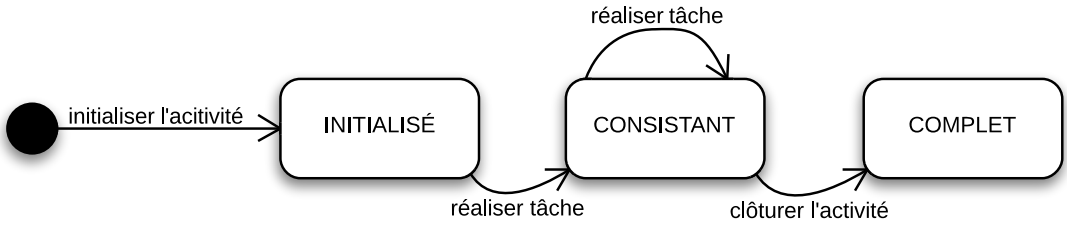


FIGURE 104 – Les états d'un modèle dans ⟨HOE⟩[2]

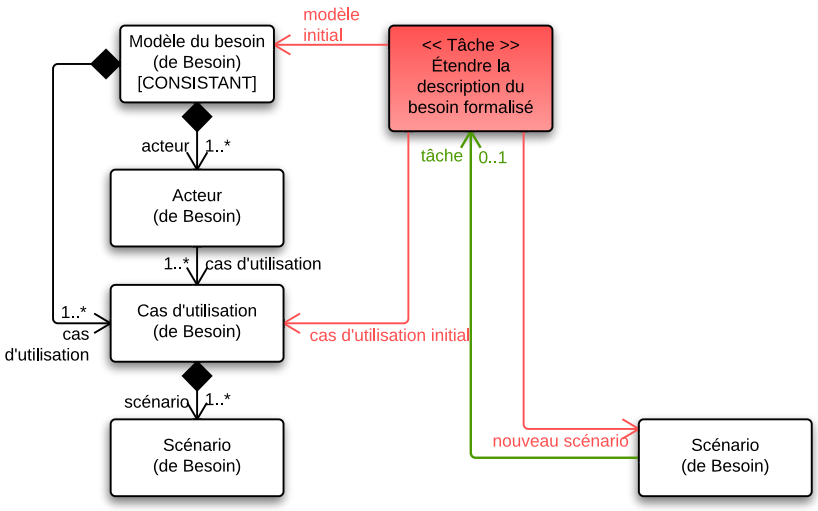
Tâche	Description
Étendre la description du besoin formalisé	Étend la description du besoin formalisé en ajoutant des scénarios à un cas d'utilisation particulier. **Réalisation :** Développeur. **Pré-condition :** modèle du besoin CONSISTANT **Post-condition :** nouveaux scénarios créés  **Explication :** Initialement, le modèle d'analyse du besoin est dans l'état CONSISTANT. Cette tâche assure la création de nouveaux scénarios pour un cas d'utilisation particulier (nommé cas d'utilisation initial). Cette tâche n'est réalisable qu'à la suite d'une première tâche d'*extension de la description du besoin modélisé*. Elle ne modifie pas l'état du modèle.

TABLE 34 – Gestion de projet : Tâche d'extension de la description du besoin modélisé

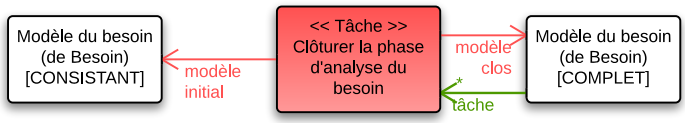
Tâche	Description
Clore la phase d'analyse du besoin	Clot le modèle du besoin une fois que ce dernier est considéré comme complet. **Réalisation :** Chef de projet (automatisable). **Pré-condition :** modèle du besoin CONSISTANT **Post-condition :** modèle du besoin COMPLET  **Explication :** Cette tâche permet de clore un modèle du besoin CONSISTANT. Cette décision est arbitraire et prise par le chef de projet lorsqu'il considère que l'ensemble du besoin formalisable du cahier des charges a été formalisé. Une fois le modèle clos, il passe à l'état COMPLET et il n'est plus possible de réaliser des tâches d'*extension* et d'*élargissement du besoin*.

TABLE 35 – Gestion de projet : Tâche de clôture de la phase d'analyse du besoin

EN RÉSUMÉ

- ✓ *Couplage entre les produits et les tâches assuré par le méta-modèle de gestion de projet*
- ✓ *Traçabilité du produit à la tâche assurée par la tâche étendant la méta-classe UML Classifier*
- ✓ *Traçabilité de la tâche au produit assurée par la tâche étendant la méta-classe UML Action*
- ✓ *Traçabilité au besoin tout au long du processus assurée par les feuilles de tâche*
- ✓ *Définition de chaque tâche en termes de réalisateur, pré et post-conditions, produits entrants et sortants*

4 Gestion de projet connectée aux modèles produits

Les sections précédentes ont permis de mettre en évidence la traçabilité bidirectionnelle entre les tâches et les produits, ainsi que la traçabilité au besoin assurée par les scénarios. Dans cette section, nous présentons comment les concepts du langage proposé dans ce chapitre permettent de concevoir une gestion de projet efficace et connectée aux modèles produits. Nous nous sommes exclusivement intéressés aux aspects de suivi et d'exécution du processus, de la gestion des ressources humaines et de la planification des itérations et des activités. Les

Id	Nom	Causalité	Description
1	S'abonner à la détection d'une présence	primaire	L'acteur s'abonne à la détection de présence. Le système le notifie alors en cas d'une détection de présence ou d'absence.
2	S'abonner au suivi d'un passant	primaire	L'acteur s'abonne au suivi d'un passant. Le système lui retourne la position d'un passant si détecté.
3	Passer en mode automatique	secondaire	L'acteur passe le système en mode automatique, ce dernier orientant automatiquement la caméra lorsqu'il détecte un visage.
4	Passer en mode manuel	secondaire	L'acteur passe le système en mode manuel, il peut alors orienter la caméra.
5	Orienter la caméra	secondaire	L'acteur oriente manuellement la camera.

Table 36 – Liste des cas d'utilisation du modèle d'analyse du besoin de l'application

Id	Nom	CU	Nature	Description
1-1	Notification de détection de présence	1	nominal	Une présence est détectée. Le système notifie de la présence détectée.
1-2	Notification de détection d'absence	1	nominal	Lorsqu'une présence initialement détectée disparait, le système notifie de l'absence de détection.
1-3	Dispositif de détection obstrué	1	erreur	Un obstacle obstrue le dispositif de détection.
2-1	Notification de suivi de passants	2	nominal	Un passant est détecté. Le système notifie de la position du passant.
2-2	Notification de suivi de passants impossible (météo)	2	erreur	Les mauvaises conditions météorologiques empêchent de poursuivre le suivi d'un passant détecté.
		...		
5-1	Orienter la camera	5	nominal	L'acteur est en mode manuel, la caméra s'oriente.
5-2	Orienter la camera en automatique	5	erreur	L'acteur est en mode automatique, donc il ne peut pas orienter la caméra.

Table 37 – Liste des scénarios des cas d'utilisation du modèle d'analyse du besoin de l'application

considérations de coûts, de risque, de disponibilité des ressources physiques, etc. n'ont pas été prises en compte mais pourraient être facilement envisagées en couplant le méta-modèle de PMBOK discuté dans la section 1. Dans le cas de cette partie, nous illustrerons la gestion de projet appliquée à notre cas d'étude, le *système de suivi de passants*.

Le processus $\langle \mathsf{HOE} \rangle^2$ est guidé par la satisfaction du besoin formalisé en termes de cas d'utilisation et de scénarios. Les tables 36 et 37 illustrent la description des différents cas d'utilisation

et scénarios du système de suivi de passants. Nous rappelons que le langage $\langle \mathsf{HOE} \rangle^2$ permet de définir une causalité (primaire ou secondaire) aux cas d'utilisation et une nature (nominal ou d'erreur). Ces propriétés permettent de définir un ordre et une priorité pour la définition des itérations et des différentes tâches de modélisation permettant de satisfaire les différents besoins formalisés sur les différentes phases du processus. Ainsi, la *détection de présence* constitue le besoin prioritaire du *système de suivi de passants*. La prise en compte d'un mode manuel et automatique, ainsi que l'orientation de la caméra est secondaire et peut être réalisé dans des itérations ultérieures du processus.

La figure 105 illustre la mise en place de la gestion de projet dans le cadre du système de suivi de passants. Elle illustre trois niveaux de modélisation.

M2. La partie haute de la figure illustre un fragment du méta-modèle des activités et de la gestion de projet dont la description complète a été réalisée dans la section 2. On y retrouve les concepts de *tâche*, *phase*, *projet*, *itération* et *feuille de tâche*.

M1. Le niveau *M1*, à gauche de la figure, illustre la modélisation sous la forme d'un diagramme d'activités d'une phase du processus $\langle \mathsf{HOE} \rangle^2$. Cette phase contient un ensemble de tâches réalisées par le chef de projet ou les développeurs. Cette modélisation correspond à la phase d'analyse du besoin et les modélisations pour les trois autres phases sont également considérées (cf. chapitre 6). La notation est celle d'un diagramme d'activités UML, mais la sémantique est spécialisée (un diagramme d'activités correspond à une phase du processus $\langle \mathsf{HOE} \rangle^2$).

M0. La partie droite de l'image représente l'exécution du processus pour le système de suivi de passants, au niveau *M0*. Deux notations sont proposées. La première reprend la notation des diagrammes d'objets UML. La seconde notation est tabulaire et est fournie pour favoriser la compréhension. Elle met en évidence le processus et les itérations dirigées par la satisfaction du besoin. Cette seconde notation ordonne les tâches de modélisation selon deux axes. L'axe horizontal définit la largeur du besoin à couvrir. Cette largeur est définie par les scénarios (notés *Sc. x*, où x est l'identifiant du scénario dans la table 37) obtenus durant la phase d'analyse du besoin. L'axe vertical correspond à la profondeur du processus, c'est-à-dire les quatre phases du processus $\langle \mathsf{HOE} \rangle^2$. Cet axe permet de définir à quel niveau de modélisation sont satisfaits les scénarios. Chaque case à la verticale d'un ou plusieurs scénarios indique la réalisation d'une tâche (notée t_x où x correspond au numéro de la feuille de tâche dans le diagramme d'objets) de modélisation réalisée. À noter que les tâches d'initialisation ou de de clôture de modèle ne satisfaisant pas de nouveaux scénarios, elles n'apparaissent pas dans la vue tabulaire.

Dans la partie suivante, nous illustrons l'organisation des feuilles de tâche selon les trois itérations et la réalisation des tâches associées. Les modèles conçus seront présentés. Lorsqu'une tâche modifie un modèle déjà existant, la modification sur le modèle est illustrée en gras.

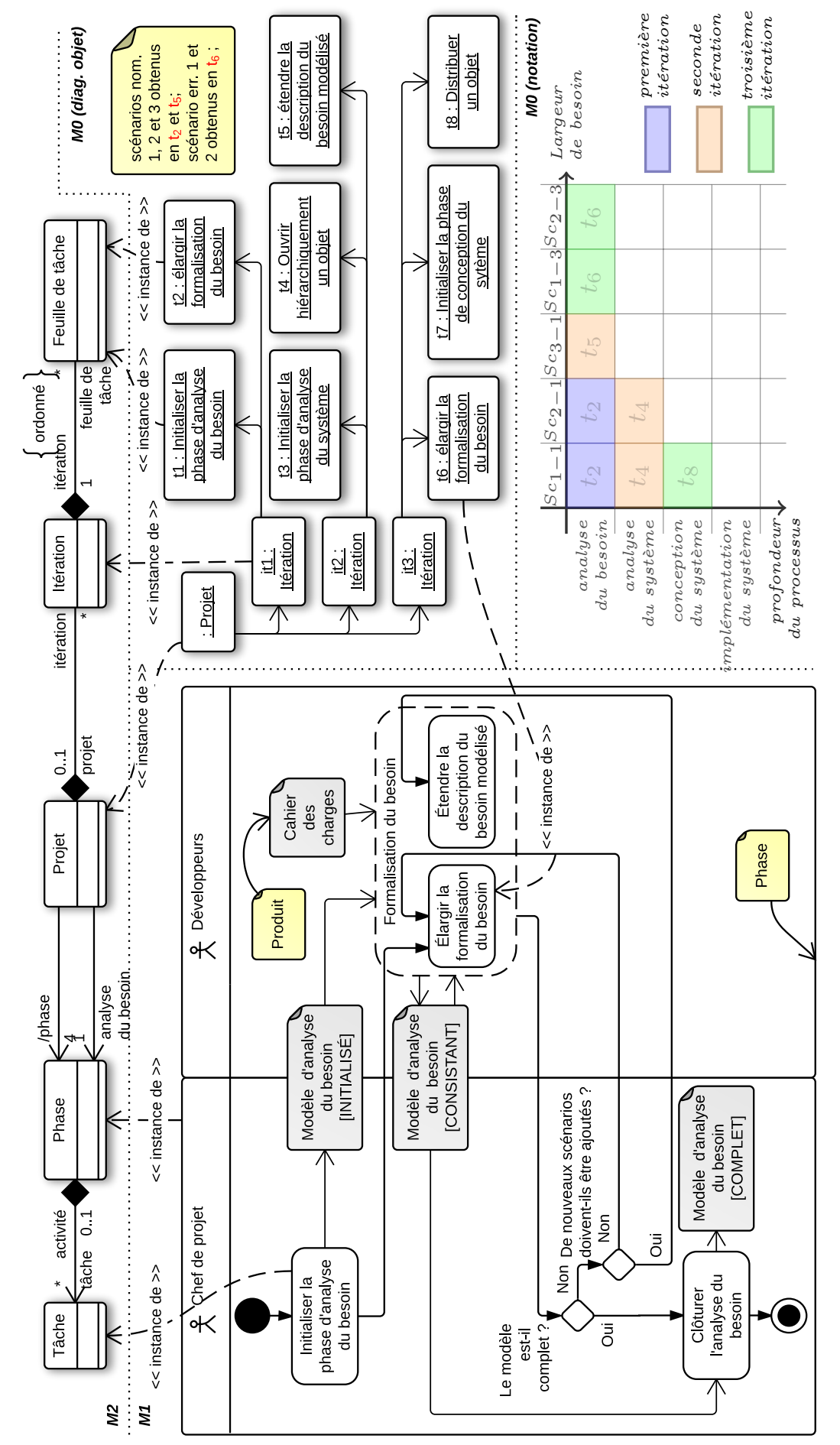


FIGURE 105 – Feuilles des tâches dans ⟨HOE⟩[2]

Itération 1 : formalisation du besoin (tâches t_1 et t_2).

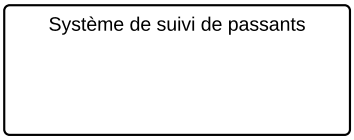
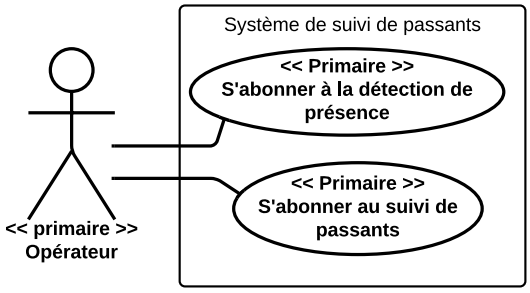
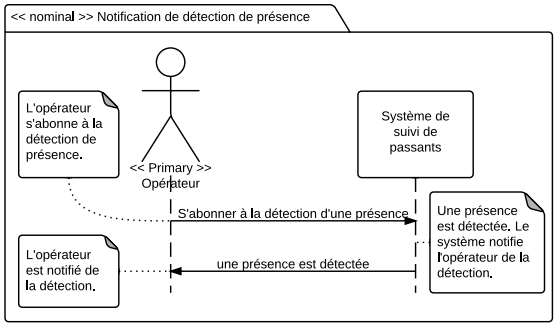


FIGURE 106 – Réalisation de t_1 : initialisation du modèle d'analyse du besoin

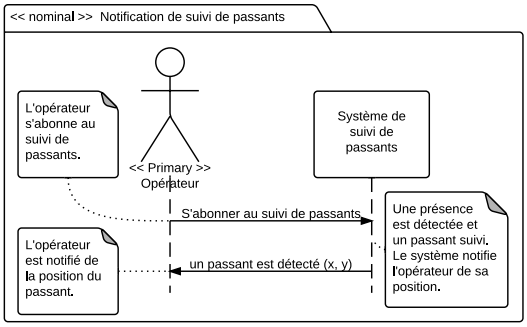
La première itération permet de définir une première formalisation du besoin du système de *suivi de passants*. La tâche t_1 permet d'*initialiser le modèle d'analyse du besoin*. Cette initialisation produit le système vide du modèle d'*analyse du besoin* illustré par la figure 106. Le modèle est alors dans l'état INITIALISÉ et nécessite une première tâche d'élargissement de la formalisation du besoin pour faire apparaitre les premiers acteurs, cas d'utilisations et scénarios. Cette tâche est réalisée dans la même itération. Il s'agit de la tâche t_2.



(a) Modélisation des deux premiers cas d'utilisation



(b) Modélisation du scénario nominal 1-1



(c) Modélisation du scénario nominal 2-1

FIGURE 107 – Réalisation de t_2 : modélisation des scénarios nominaux 1-1 et 2-1

La tâche t_2 d'*élargissement de la formalisation du besoin* permet d'identifier l'acteur du système (l'*Opérateur*), les deux premiers cas d'utilisation (cf. table 36) et les scénarios nominaux *1-1* et *2-1*. Son résultat est illustré par la figure 107. Dans cette figure, nous avons modélisé les deux cas d'utilisation du système initial (cf. fig. 107a) et nous avons ensuite modélisé un scénario nominal pour chacun (cf. fig. 107b et 107c). Le modèle d'*analyse du besoin* devient CONSISTANT. La tâche t_2 conclut la première itération. Il est désormais possible de poursuivre la formalisation du besoin ainsi que d'initialiser la phase d'analyse du système.

Itération 2 : formalisation du besoin (tâche t_5) et ouverture du système (tâches t_3 et t_4).

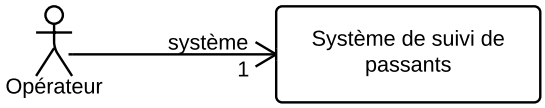


FIGURE 108 – Réalisation de t_3 : initialisation du modèle d'analyse du système

L'objectif de la seconde itération est double. Elle permet de descendre dans les phases du processus afin de satisfaire les scénarios prioritaires *1-1* et *2-1* en phase d'analyse du système. La première tâche (t_3) consiste à *initialiser la phase d'analyse du système*. Cette tâche est illustrée par la figure 108 qui présente le modèle d'*analyse du système* dans l'état INITIALISÉ.

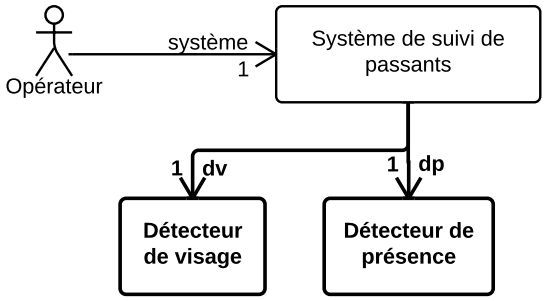
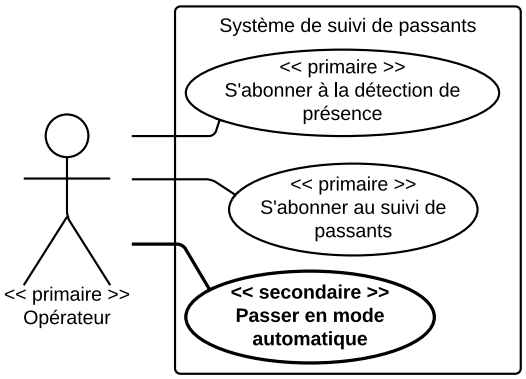
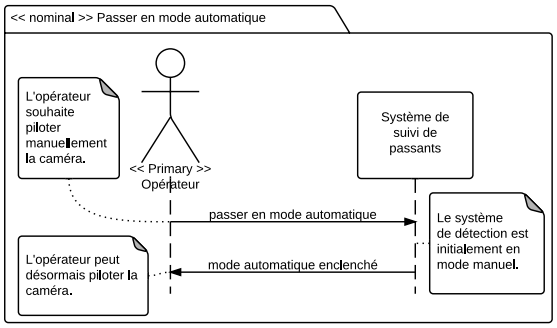


FIGURE 109 – Réalisation de t_4 : satisfaction des scénarios 1-1 et 2-1 en analyse du système

La tâche t_4 permet de satisfaire les deux scénarios *1-1* et *2-1* en phase d'analyse du système. Cette tâche est illustrée par la figure 109. Afin de les satisfaire, le développeur modélise un détecteur de présence et un détecteur de visage dans le modèle d'analyse du système de suivi de passants durant la tâche t_4 (cf. fig. 109). La tâche t_5 étend la formalisation du besoin en identifiant le troisième cas d'utilisation (cf. fig. 110) et le scénario nominal de ce dernier.



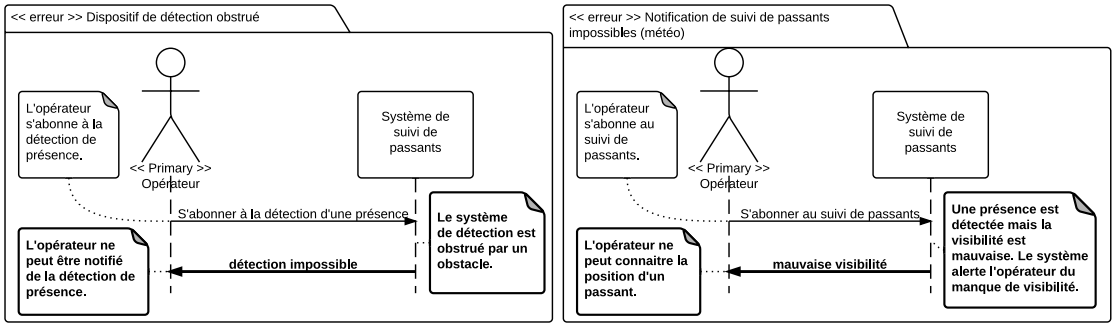
(a) Modélisation du cas d'utilisation 3



(b) Modélisation du scénario nominal 3-1

FIGURE 110 – Réalisation de t_5 : modélisation du scénario nominal 3-1 en analyse du besoin

Itération 3 : formalisation du besoin (tâche t_6) et distribution (tâches t_7 et t_8).



(a) Modélisation du scénario d'erreur 1-3 (b) Modélisation du scénario d'erreur 2-2

FIGURE 111 – Réalisation de t_6 : modélisation des scénarios d'erreur 1-3 et 2-1

La troisième itération permet de continuer à formaliser l'analyse du besoin en identifiant deux scénarios d'erreur des cas d'utilisation primaires *1* et *2*. Dans le cas de la détection de présence, le scénario d'erreur *1-3* (cf. fig. 111a) prévient en cas d'obstruction du dispositif de détection de présence. Le scénario d'erreur *2-2* (cf. fig. 111b) permet quant à lui de notifier l'utilisateur de mauvaises conditions météorologiques empêchant le suivi d'un passant. La tâche t_6 permet la modélisation de ces deux scénarios (cf. fig. 111).

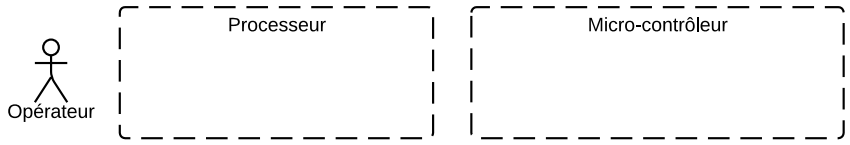


FIGURE 112 – Réalisation de t_7 : initialisation du modèle de conception du système

Enfin, les tâches t_7 (cf. fig. 112) et t_8 (cf. fig. 113) permettent de d'initialiser et d'entamer la phase de conception du système en satisfaisant le premier scénario par un découpage des objets sur les différents mondes de la plate-forme.

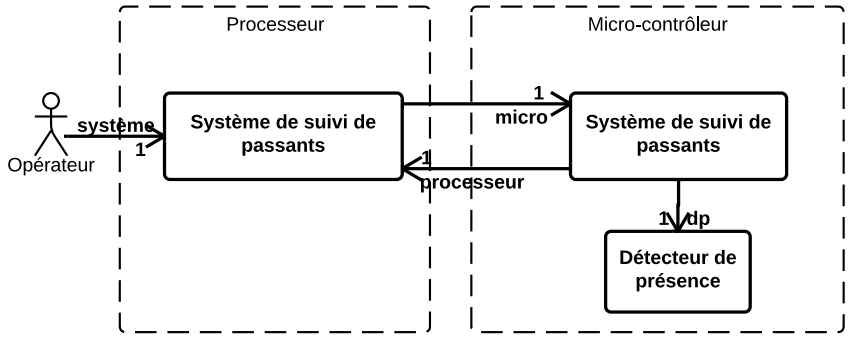


FIGURE 113 – Réalisation de t_8 : satisfaction du scénario 1-1 en phase de conception du système

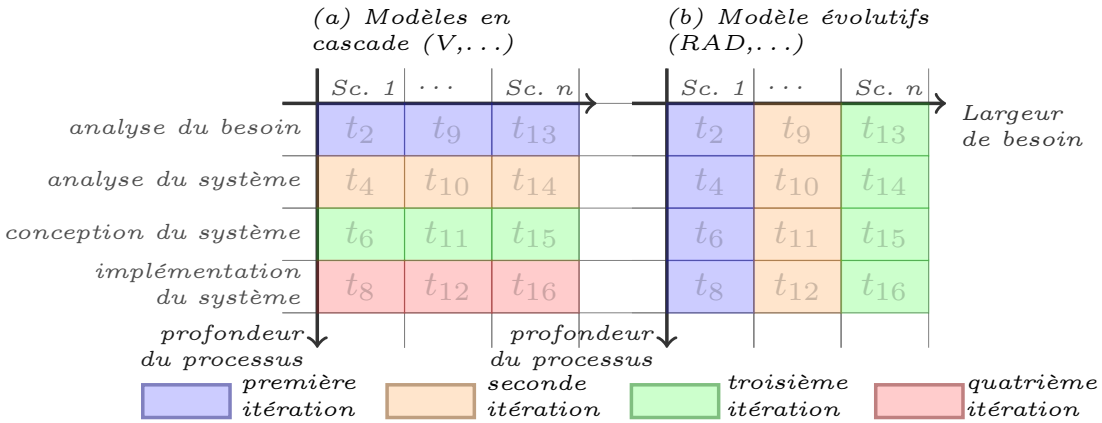


FIGURE 114 – Organisation des itérations

L'exemple présenté illustre plusieurs bénéfices de la gestion de projet. La planification des tâches est facilitée par l'utilisation de la notation graphique. Le chef de projet peut opter pour une couverture du processus *en profondeur*, en sélectionnant un ensemble de scénarios prioritaires (fig. 114 à droite). Il peut sinon opter pour une couverture *en largeur* en choisissant de formaliser tout le besoin avant d'entamer les phases suivantes (fig. 114 à gauche). La définition des itérations permet au chef de projet d'établir un *compromis dans l'organisation des tâches et la définition des activités pour s'adapter à tous les types de projet.*

Le second bénéfice se situe au niveau de la *parallélisation* des tâches. La notation choisie pour l'exécution ne laisse pas entrevoir les dépendances entre les différentes tâches, mais il est facile de les expliciter. Ce faisant, il serait possible de voir quelles tâches sont dépendantes et ne peuvent être réalisées en parallèle, et lesquelles ne sont pas dépendantes et peuvent l'être. Pour le *système de suivi de passants* par exemple, il n'est pas possible de réaliser la tâche t_8 de distribution sur les mondes de la plate-forme avant d'avoir initialisé le modèle durant la tâche t_7. néanmoins, il est toujours possible de poursuivre la formalisation du besoin en même temps.

Enfin, l'exemple présenté permet de fournir au chef de projet un moyen pour *mesurer le temps de développement et organiser son équipe.* L'utilisation des propriétés (cf. fig. 103) pour dater les feuilles de tâches en cours de réalisation nous permet de connaitre les charges de travail de chaque développeur. Bien que non abordé dans nos travaux, cet aspect est facile à prendre en compte et offrirait au chef de projet des moyens de gérer efficacement les différents membres de son équipe.

EN RÉSUMÉ

- ✓ *Couplage de la gestion de projet aux modèles produits*
- ✓ *Proposition d'une notation graphique pour suivre l'exécution des tâches (M0)*

Synthèse du chapitre

Dans ce chapitre, nous avons présenté le méta-modèle de la gestion de projet $\langle \mathsf{HOE} \rangle^2$. Ce méta-modèle favorise une gestion de projet efficace et fortement connectée aux produits développés, de manière à ce que, à tout instant dans le processus, il soit possible de tracer les produits développés et de connaitre les raisons de leur développement. L'efficacité de cette gestion de projet est due à l'établissement d'une traçabilité entre le produit développé et la feuille de tâche définissant les scénarios à satisfaire. Ce méta-modèle met également en évidence des concepts basiques tels que le *projet* ou l'*itération* permettant d'offrir au chef de projet un moyen efficace de mesurer et piloter la progression, d'assigner des tâches, et établir une stratégie de développement.

En dernier lieu, il est nécessaire de rappeler qu'une gestion de projet efficace doit être considérée au niveau de son outillage. Cet outillage doit favoriser une ingénierie efficace et offrir un ensemble de fonctionnalités permettant de visionner, de mesurer, d'établir des stratégies et de piloter la gestion de projet. Nous avons intégré une preuve de concept de l'intégration de la gestion de projet dans l'outillage *CanHOE2*, l'outillage dédié à la méthode $\langle \mathsf{HOE} \rangle^2$. Cet outil fait l'objet du chapitre suivant.

Chapitre 9

CanHOE2, un atelier de développement dédié

Publication pertinente à ce chapitre :

[**Hili *et al.* 2014a**] Nicolas Hili, Christian Fabre, Sophie Dupuy-Chessa et Dominique Rieu. *A Model-Driven Approach for Embedded System Prototyping and Design.* In IEEE International Symposium on Rapid System Prototyping (RSP'14) (part of ESWEEK'14), New Dehli, India, Octobre 2014

Liste des sections

Dans la section 3, nous évoquions la nécessité de posséder des outils connectés et dédiés. L'outillage est une partie essentielle pour permettre l'utilisation des méthodes de développement. Il doit guider le processus, favoriser un développement collaboratif et améliorer grandement le développement par l'automatisation de certaines activités, la génération de code et de documentation.

Nous introduisons CanHOE2, un outillage dédié au support de la méthode $\langle \mathsf{HOE} \rangle^2$ pour le développement des systèmes embarqués. CanHOE2 signifie *CANonical* $\langle \mathit{HOE} \rangle^2$. Il est canonique au sens où il est conçu pour le support de la méthode $\langle \mathsf{HOE} \rangle^2$. Nous avons développé l'outillage en accord avec les quatre axes identifiés dans la section 1 : *support au langage*, *support au processus*, *support à la gestion de projet* et *support à la gestion de versions*. Un effort important a été réalisé sur le support au langage avec la conception d'éditeurs graphiques *dédiés* au développement des modèles $\langle \mathsf{HOE} \rangle^2$. Cette contribution a été présentée lors de la *Eclipse DemoCamps* à Grenoble [Hili 2013a]. Le support à la gestion de projet et au processus collaboratif ont été étudiés dans un dernier temps et brièvement discutés dans [Hili *et al.* 2014a].

Ce chapitre est structuré de la façon suivante : la section 1 introduit et détaille les principales caractéristiques de l'outil CanHOE2 ; la section 2 présente l'outil sous le regard de la conception des modèles $\langle \mathsf{HOE} \rangle^2$; la section 3 identifie les éléments mis en place pour le support du processus ; la section 4 aborde la prise en compte de la gestion de projet ; enfin, la section 5 décrit le support à la gestion de versions.

1 Cadre général et orientation du développement

Nous avons fait le choix de développer un outil dédié. Cet outil a pour objectif de supporter la méthode $\langle \mathsf{HOE} \rangle^2$ dans tous ses aspects et se limite à cette dernière. Nous ne nous sommes pas intéressés au développement d'un outil industrialisable – la durée de la thèse ne permettant pas la production d'un tel outil – aussi nous avons laissé de côté tous les critères d'ergonomie et d'usabilité et nous nous sommes restreints à une définition très simple et claire de l'interface. Dans cette section, nous commençons par exprimer nos besoins fonctionnels et non fonctionnels. Nous présentons ensuite l'environnement de développement choisi et détaillons nos choix d'orientation du développement.

1.1 Expression des besoins fonctionnels

La figure 115 illustre les différents besoins qui doivent être satisfaits par l'outil CanHOE2. Nous avons identifié deux acteurs, le chef de projet et le développeur, ainsi que six besoins fonctionnels que doit satisfaire l'outil CanHOE2.

Concevoir graphiquement des modèles Le langage $\langle \mathsf{HOE} \rangle^2$ procure un certain nombre de modèles qu'il faut pouvoir éditer et visualiser. Ces modèles sont conformes à la définition du méta-modèle $\langle \mathsf{HOE} \rangle^2$, ce dernier étant une spécialisation du langage UML. L'environnement de développement doit nous permettre de définir notre méta-modèle comme une spécialisation d'UML.

Transformer des modèles L'usage d'éditeurs graphiques nous permet de concevoir les modèles selon le méta-modèle $\langle \mathsf{HOE} \rangle^2$, présenté dans le chapitre 6. Le méta-modèle $\langle \mathsf{HOE} \rangle^2$ étant structuré en plusieurs parties, et le processus $\langle \mathsf{HOE} \rangle^2$ en plusieurs phases, l'environnement de développement doit nous fournir des outils pour transformer les modèles produits. Ces aspects sont utiles pour le raffinement des modèles d'une phase à une autre, ainsi que pour l'injection dans les conteneurs et l'exécution des règles d'implémentation.

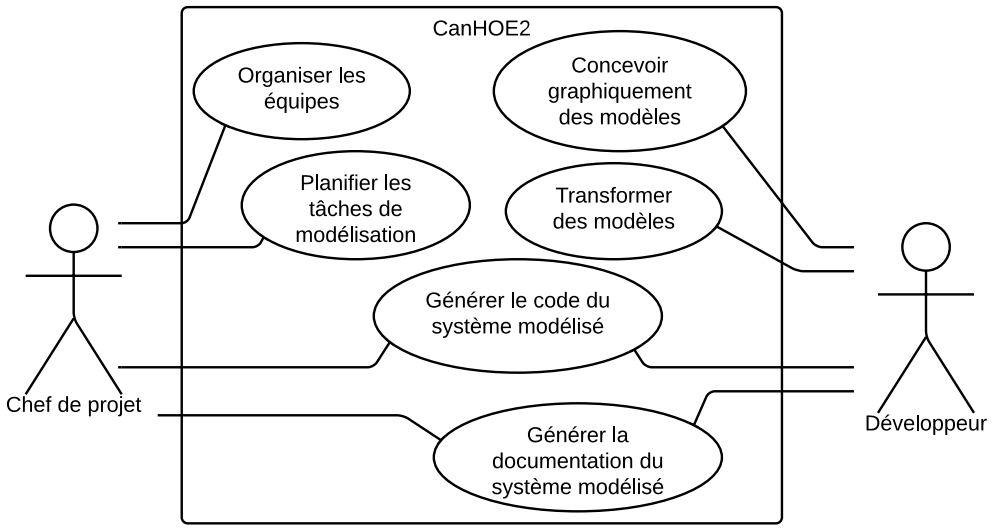


FIGURE 115 – Fonctionnalités de l'outil dédié

Générer la documentation du système modélisé. Afin de répondre aux problématiques de certification et de documentation des systèmes embarqués identifiées dans l'état de l'art, il est important de pouvoir générer de la documentation sur les modèles développés, de documenter la traçabilité entre modèles, la traçabilité aux besoins, ainsi que l'enchainement des tâches ayant mené à ces modèles.

Générer le code du système modélisé. La génération de code sur la plate-forme matérielle finale constitue le principal objectif de $\langle \mathsf{HOE} \rangle^2$. Mais il est également important, tant que faire se peut, de pouvoir générer du code intermédiaire pour exécuter ou simuler les modèles produits à chaque étape du processus.

Organiser les équipes. L'outil CanHOE2 doit supporter la gestion de projet telle que nous l'avons présentée dans le chapitre 8. Pour cela, il doit permettre à un chef de projet d'organiser ses équipes.

Planifier les tâches de modélisation. La planification des tâches est le second aspect de la gestion de projet que l'outil doit supporter. Il doit assister le chef de projet dans la création, l'assignation et planification des tâches de développement aux différents développeurs.

1.2 Expressions des besoins non fonctionnelles

En plus des six besoins fonctionnels identifiés dans la partie précédente, nous avons identifié trois besoins non fonctionnels de l'outil.

Gérer plusieurs utilisateurs et rôles. Nous avions résumé les rôles des participants dans la méthode $\langle \mathsf{HOE} \rangle^2$ aux deux fondamentaux, le *développeur* et le *chef de projet*. L'environnement de développement doit être similaire pour tous les rôles du processus, tout en étant personnalisé

pour chacun. Ainsi, l'outil doit permettre d'identifier le rôle d'un utilisateur vis-à-vis d'un projet et de personnaliser son environnement de développement, avec des vues de gestion de projet pour le chef de projet et des éditeurs pour le développeur.

Gérer les versions des modèles dans un dépôt collaboratif. L'environnement de développement devant être multi-utilisateurs et favoriser le travail collaboratif et en parallèle, l'outil doit permettre de gérer les versions des modèles dans un dépôt de gestion de versions collaboratif. La gestion doit s'effectuer de façon transparente à l'utilisateur.

Permettre un développement multi plates-formes. L'outil CanHOE2 développé ne doit pas être contraint à un système d'exploitation. Les différents intervenants dans le processus de développement de systèmes embarqués n'utilisent couramment pas les mêmes systèmes. Des environnements Linux correspondent plus souvent aux habitudes de développeurs, tandis que le système d'exploitation Windows est plus traditionnellement utilisé par les chefs de projets. Ainsi, l'outil développé doit pouvoir être facilement déployable et installable sur l'ensemble de ces environnements.

1.3 La plate-forme Eclipse

La plate-forme Eclipse permet de répondre à différents besoins évoqués ci-dessus : environnement personnalisé, conception graphique et transformation de modèles, etc. Il est basé sur Equinox, une implémentation du standard OSGi. Il permet le développement d'extensions dans le langage Java, pour adapter la plate-forme à nos besoins. Eclipse est un environnement libre [1], en constante évolution et possédant une communauté très réactive. En outre, le code développé est multi plates-formes et très facilement déployable aux moyens des outils Eclipse.

Un environnement évolutif. Eclipse permet le développement de *greffons* [2]. le concept de *greffons* correspond à celui de *bundle* de OSGi. Un greffon permet de concevoir une *extension* de l'environnement Eclipse. Chaque greffon est défini par un fichier *Manifest* le décrivant ainsi que ses dépendances avec d'autres greffons. Eclipse est à cet effet un environnement modulaire évolutif dans lequel des greffons de différents éditeurs peuvent être conçus et déployés.

Une interface modulaire. La figure 116 illustre la constitution d'une fenêtre dans Eclipse. Cette dernière est constituée d'une zone centrale, dans lequel il est possible d'ajouter des éditeurs Eclipse. Différentes vues peuvent être ajoutées. Par exemple, dans la figure 116, il y a deux vues à gauche de la zone d'édition, le navigateur de projet et une vue d'ensemble. Ces vues sont organisées par onglet et peuvent être installées tout autour de la zone d'édition [3].

Dans Eclipse, éditeurs et vues sont agencés dans des *perspectives* [Springgay 2001]. Chaque perspective permet d'adapter l'environnement à un usage ou un outil particulier. Par exemple, la perspective Java permet le développement de programmes Java. La perspective Papyrus MDT

1. Eclipse s'appuie sur la licence non restrictive Eclipse Public License (EPL).
2. En anglais, plug-in.
3. Eclipse 3.x distinguait réellement les vues des éditeurs, qui ne pouvaient alors pas se mélanger. Depuis Eclipse 4.x, cette distinction tend à disparaitre et il est désormais possible de mélanger les vues avec les éditeurs dans une fenêtre.

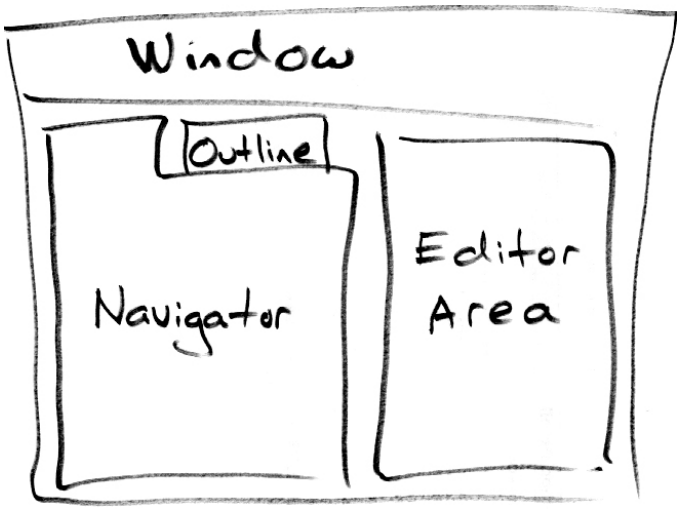


FIGURE 116 – Constitution d'une fenêtre dans Eclipse [Springgay 2001]

(cf. fig. 117) permet la manipulation de modèles UML au sein de l'environnement graphique Papyrus MDT.

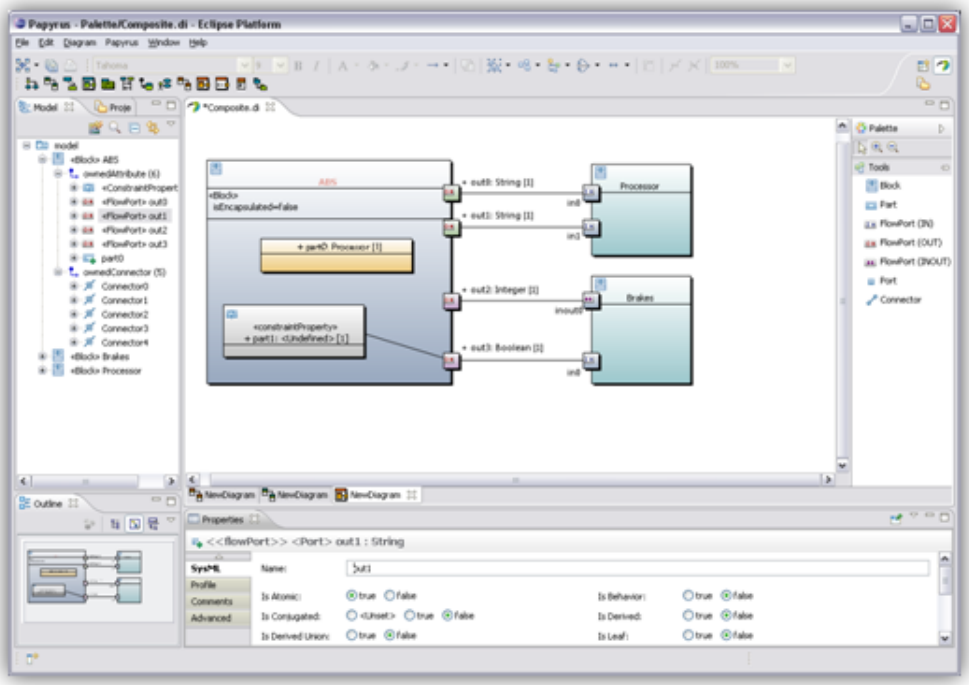


FIGURE 117 – Perspective Papyrus MDT [Papyrus 2014]

Un environnement pour la modélisation. Une déclinaison de l'environnement Eclipse, nommée Eclipse Model Development Tools (MDT), inclut un ensemble d'outils pour la conception, l'édition et la manipulation de modèles. Cet environnement est basé sur le méta-modèle Ecore, une implémentation du MOF. La définition de ce méta-modèle est une force de la plateforme Eclipse MDT et favorise une interopérabilité entre tous les outils dédiés à la manipulation des modèles dans Eclipse. Ces outils, basés sur Ecore, offrent un ensemble d'opérations sur les modèles : méta-modélisation, modélisation, transformation de modèles, migration, génération de code, etc. Ils reposent sur le projet Eclipse Modeling Framework (EMF), qui fournit une librairie facilitant le développement et la manipulation de modèles au format Ecore dans Eclipse, ainsi que quelques outils (éditeurs arborescents, validateurs, etc.) pour les manipuler.

L'évolutivité de l'environnement, l'interface modulaire et son intégration d'outils de modélisation font d'Eclipse un environnement adapté pour le développement d'un outil dédié à la méthode $\langle\mathsf{HOE}\rangle^2$. Le développement de greffons permet de contribuer aux différentes perspectives, vues et éditeurs présents dans l'environnement Eclipse, ou d'en créer de nouveaux afin de dédier l'outil à la méthode. Les zones d'édition et de visualisation permettent d'ajouter des vues et des éditeurs graphiques pour la conception des modèles. Ces éditeurs graphiques manipulent des modèles développés au format Ecore, permettant d'utiliser les divers outils de modélisation existants sur la plate-forme. En outre, la gestion de perspectives permet d'adapter l'éditeur en fonction de l'utilisateur (développeur ou chef de projet) et ainsi d'incorporer une gestion multi-utilisateurs et une gestion de projet intégrée.

1.4 Organisation et orientation du développement

Le développement a été organisé en greffons. La figure 118 et le tableau 38 illustrent les greffons organisés selon six catégories. Nous avons orienté le développement de l'outil CanHOE2 pour le support du langage, du processus, à la gestion de projet et à la gestion de version. Différentes personnes ont contribué à son développement. J'ai mis en place les briques élémentaires pour la conception des éditeurs graphiques de la méthode $\langle\mathsf{HOE}\rangle^2$. J'ai implémenté les modèles des différentes phases du processus et ai partiellement réalisé les éditeurs graphiques des phases d'analyse du besoin et du système. J'ai également développé les générateurs de code permettant de générer la documentation à partir des modèles $\langle\mathsf{HOE}\rangle^2$ et le code sur diverses plates-formes. Enfin, j'ai implémenté des règles de transformation de modèles afin de réaliser les transformations lors de l'activité d'injection dans les conteneurs de la phase d'implémentation du système. Anthony Gauchy, étudiant à l'*Institut universitaire de technologie Grenoble 2* a poursuivi le développement des éditeurs graphiques des modèles de la phase d'analyse du besoin $\langle\mathsf{HOE}\rangle^2$ et mis en place les briques élémentaires de gestion de projet, durant son stage de fin d'études (2 mois) au sein de l'équipe *Systèmes d'Information - inGénierie et Modélisation Adaptables (SIGMA)* du *Laboratoire d'informatique de Grenoble*. Yassine Ben Atitallah, étudiant à l'*École nationale des sciences de l'informatique* (ENSI, Tunisie) a poursuivi le développement des éditeurs graphiques des modèles de la seconde phase du processus $\langle\mathsf{HOE}\rangle^2$, durant son stage de fin d'études d'ingénieur (6 mois) au sein du *Laboratoire infrastructure et atelier logiciel pour puces (LIALP)* du Commissariat à l'énergie atomique et aux énergies alternatives (CEA). Le développement de l'outil est depuis repris par Fayçal Benaziz, apprenti de l'*École nationale supérieure d'informatique et de mathématiques appliquées* (ENSIMAG, Grenoble) depuis septembre 2013

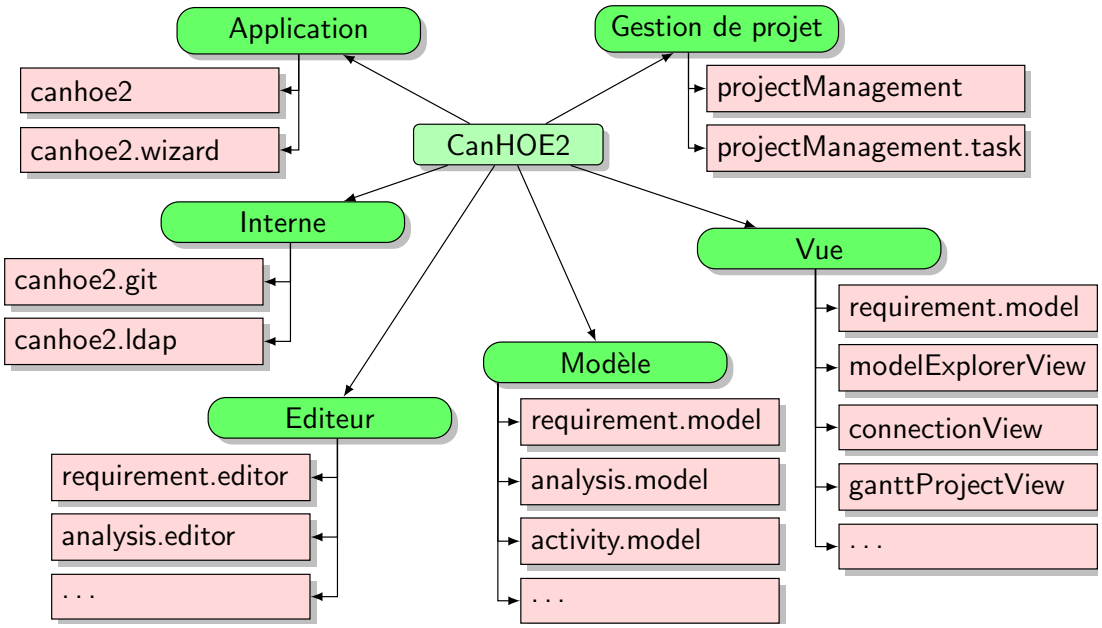


FIGURE 118 – Organisation des différents greffons

Catérogie	Description	Nombre de greffons
Application	Contient les greffons permettant la construction de l'outil dédié basé sur la plate-forme Eclipse et en tant que produit unique.	2
Interne	Contient les greffons internes à l'outil CanHOE2, exportant des paquetages pour la connexion à différents protocoles (Git, Lightweight Directory Access Protocol (LDAP)).	2
Éditeur	Contient les greffons des éditeurs et des diagrammes de chaque phase.	7
Modèle	Contient les greffons de l'implémentation des méta-modèles ⟨HOE⟩^{2} et le code généré.	9
Vue	Contient les greffons ajoutant des vues à la plate-forme Eclipse.	8
Gestion de projet	Contient des greffons spécifiques à la gestion de projet.	2

TABLE 38 – Organisation des greffons

jusqu'à septembre 2016. Benaziz contribue aujourd'hui aux développements de la gestion de projet intégrée et au suivi du processus ⟨HOE⟩^{2}.

En résumé :

- ✓ *Développement au sein de la plate-forme Eclipse*
- ✓ *Développement sous formes de greffons*
- ✓ *Orientation du développement pour le support du langage, du processus, à la gestion de projet et à la gestion de versions*

2 Support au langage

Afin de supporter efficacement la méthode ⟨HOE⟩^{2}, l'outil CanHOE2 doit supporter son langage, défini par quatre méta-modèles correspondant aux quatre phases. Dans cette section, nous présentons l'intégration des méta-modèles ⟨HOE⟩^{2} dans l'environnement de développement Eclipse et le développement des éditeurs graphiques pour la conception de ces modèles.

2.1 Rappel de nos besoins

Nous avons identifié différents besoins pour le développement des éditeurs graphiques des modèles ⟨HOE⟩^{2}. Tout d'abord, les méta-modèles ayant été définis comme extension du méta-modèle UML et saisis sur la base du projet UML2, ils possèdent à ce titre une modélisation basée sur EMF et Ecore. Ainsi, les éditeurs graphiques doivent permettre de concevoir des modèles EMF, et plus particulièrement des modèles UML.

La syntaxe concrète du langage ⟨HOE⟩^{2} est très proche de celle d'UML, ce qui nous permettrait de réutiliser une partie des diagrammes UML (diagramme de classes, de séquences, de machines à états et d'activités). Il y aurait donc un fort intérêt à réutiliser et étendre des éditeurs graphiques UML existants dans la plate-forme Eclipse, sous condition de pouvoir les adapter au langage ⟨HOE⟩^{2}. À défaut d'éditeurs UML existants ou d'impossibilité de les adapter à la notation de ⟨HOE⟩^{2}, il doit être possible de concevoir nos propres diagrammes ⟨HOE⟩^{2}.

L'outil de conception des éditeurs graphiques doit donc répondre à plusieurs critères : supporter les modèles EMF et UML implémentés en Ecore ; permettre d'adapter le rendu visuel des concepts modélisés afin de correspondre le plus fidèlement possible à la notation du langage ⟨HOE⟩^{2}. Enfin, l'outil doit permettre un développement collaboratif, itératif et incrémental. Nous avons pour cela testé trois outils, GMF, Papyrus MDT et Graphiti.

2.2 Études des solutions et des outils existants

Dans cette partie, nous étudions et comparons quelques outils, selon deux perspectives : la conception des méta-modèles du langage ⟨HOE⟩^{2} et le développement des éditeurs graphiques.

Le projet UML2. Le projet UML2, présent dans l'environnement Eclipse MDT, permet de développer des modèles conformes aux dernières versions du langage UML. Ce projet offre une implémentation au format Ecore du méta-modèle UML, de sorte qu'il soit possible de concevoir

des modèles UML, tout en pouvant les manipuler dans tous les autres outils de l'environnement Eclipse MDT : génération de code avec les outils Acceleo / Xtend, transformation de modèles avec Atlas Transformation Language (ATL) ou Eclipse Transformation Language (ETL), éditeurs graphiques avec Papyrus MDT, Graphiti ou GMF, etc.

En plus de la définition des méta-modèles UML au format Ecore, le projet UML2 met à disposition un éditeur arborescent de modèles basé sur EMF. Cet éditeur est illustré par la figure 120. Dans cet éditeur, totalement compatible avec les dernières versions du standard, il est possible de concevoir différents modèles UML, mais aussi de définir ou d'appliquer des profils, ou encore *concevoir des extensions du méta-modèle UML.*

L'outil Papyrus MDT. Papyrus MDT est un outil de conception de modèles UML développé au CEA LIST. Il est basé sur le projet UML2 et sur GMF (qui sera présenté plus loin) qui a été utilisé pour produire le code des différents diagrammes. Le code généré a été adapté pour obtenir des diagrammes UML avec un rendu visuel respectant la notation UML et un comportement intuitif pour la modélisation et la manipulation des modèles. Papyrus MDT fournit un multi-éditeur (dans Eclipse, un éditeur contenant d'autres éditeurs et permettant leur instanciation et leur organisation au moyen d'onglets) pour embarquer les diagrammes UML, une palette à outils, un menu de propriétés et un explorateur de modèles.

Papyrus MDT permet la définition et l'exportation de profils UML, la personnalisation (simple) des éditeurs UML, de la palette, du menu de propriétés et de l'explorateur de modèles lors de l'application d'un profil, ainsi que la *définition d'extensions du méta-modèle UML.* Papyrus MDT est également adapté pour la *conception d'éditeurs graphiques*, ou plus justement l'adaptation des diagrammes UML existants. L'adaptation des éditeurs est simple (bien que limitée) et déclarative dans le cas d'une approche par profilage UML, générative, complexes (et non documentée) dans le cas d'une approche par extension du méta-modèle UML.

Afin d'adapter les diagrammes actuels, il est nécessaire de regénérer les diagrammes à partir de GMF et de les personnaliser avec les nouveaux concepts de notre extension. Puisque Papyrus MDT ne fournit aucun assistant d'adaptation, il est nécessaire de modifier le code généré à la main afin de satisfaire notre personnalisation. Cette approche s'est avérée très complexe et nous a dissuadés de l'utiliser pour concevoir les diagrammes $\langle\mathsf{HOE}\rangle^2$ inspirés des diagrammes UML. Plus récemment, de nouvelles fonctionnalités ont été ajoutées à Papyrus MDT, telles que l'usage de feuilles de style CSS [4], des assistants pour la personnalisation des menus de propriétés, de la génération de code. Nous n'avons cependant pas pu toutes les tester.

Le projet GMF. GMF est un projet Eclipse permettant de *concevoir des éditeurs graphiques* basés sur la définition de méta-modèles au format Ecore. Il est composé de deux parties : *GMF runtime*, une infrastructure offrant un support pour la création d'éditeurs dans Eclipse, et *GMF tooling* qui propose une approche générative (cf. fig. 119), un ensemble d'outils et de modèles GMF permettant de générer le code d'éditeurs graphiques (cette approche est celle utilisée dans Papyrus MDT). Dans cette approche, des modèles GMF [5] sont successivement créés et raffinés à partir du méta-modèle. GMF met à disposition un tableau de bord, illustré par la figure 119,

4. En anglais, Cascading Style Sheet (CSS).

5. *Graphical Def Model*, *Tooling Def Model*, *Mapping Model* et *Diagram Editor Gen Model* sont des modèles GMF pour générer des éditeurs graphiques, à différencier des modèles $\langle\mathsf{HOE}\rangle^2$ conformes aux méta-modèles présentés auparavant. Pour éviter toute confusion possible, nous expliciterons constamment *modèles GMF ou modèles* $\langle\mathit{HOE}\rangle^2$.

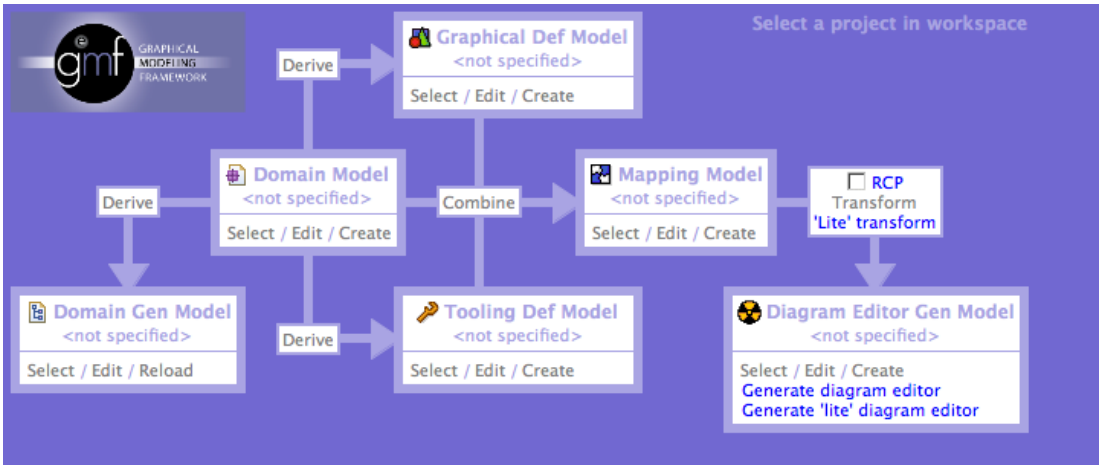


FIGURE 119 – Tableau de bord de l'outil GMF

pour la conception des différents modèles GMF. Il permet d'avoir une vision globale des modèles GMF créés et restant à créer. Différentes actions sont possibles afin de dériver ou de combiner des modèles GMF. Chaque modèle GMF créé offre des informations supplémentaires utilisées pour générer un éditeur graphique. Par exemple, le modèle GMF nommé *Graphical Def Model* définit différents rendus visuels, tandis que le modèle GMF *Tooling Def Model* permet de définir les différents outils employés dans l'éditeur pour la création et l'édition (élément dans la palette graphique de l'éditeur, menu contextuel, etc.) des éléments d'un modèle $\langle \mathsf{HOE} \rangle^2$. Le modèle GMF d'association *Mapping Model* est la combinaison des trois modèles GMF précédents et permet d'associer pour chaque concept d'un méta-modèle $\langle \mathsf{HOE} \rangle^2$ un rendu visuel et des outils. Le dernier modèle GMF permet de générer le code Java de l'éditeur graphique.

Cette approche est totalement générative et incrémentale. Il est possible de générer le code par itérations successives pour couvrir progressivement l'ensemble des concepts du méta-modèle. Cependant, cette approche est extrêmement limitée pour adapter le comportement de l'éditeur modélisé dont le code généré nécessite très souvent d'être retouché à la main. La complexité du code source généré accentue grandement la difficulté de cette phase de développement, et le code retouché présente un grand risque d'empêcher de futures générations de code. Enfin, le rendu visuel est très limité, très souvent à l'expression de « boites et flèches » et des rendus plus complexes rendent la modification du code à la main inévitables.

Graphiti. Graphiti est un *outil de conception d'éditeurs graphiques* basé sur Ecore. Il ne cible pas directement la modélisation UML, mais peut s'appuyer dessus, puisque le projet UML2 a défini une version du méta-modèle UML implémentée en Ecore. Graphiti ne suit pas une approche générative ou déclarative comme ses deux prédécesseurs. L'ensemble du code de l'éditeur est écrit à la main – il est toutefois possible d'adopter une approche générative en utilisant l'outil Spray permettant de générer, à partir d'un langage dédié, des éditeurs Graphiti. Graphiti fournit une librairie très soignée permettant le développement d'éditeurs graphiques très simplement, tout en permettant un très haut niveau de personnalisation.

Graphiti facilite très fortement un développement incrémental, où chaque incrément cor-

Caractéristiques / outil	Papyrus MDT	UML2
Éditeurs		
Type	graphique	arborescent
Assistants	◐	◐
Fonctionnalités de l'éditeur		
Création de méta-classes	◐	●
Création d'associations	●	◐
Généralisation des concepts UML	◐	●
Redéfinition de propriétés UML	○	●
Contraintes OCL	●	●
Communauté & Développement		
Documentation (forum, wiki, etc.)	○	●
Évolution	◐	◐

Table 39 – Extension du méta-modèle UML : les outils Papyrus MDT et UML2

respond à une nouvelle fonctionnalité ajoutée pour un concept donné. Le concept fondamental de Graphiti est celui de *fonction* (*feature*), permettant de décrire un comportement simple (ajout, cliquer-déplacer, suppression, effacement, déplacement) ou complexe (modification du rendu visuel, édition directe de texte, connexion, etc.) utilisable par un ou plusieurs concepts du méta-modèle. Une fonction est distribuée par un fournisseur de fonctions (*featureProvider*) en fonction du *contexte d'exécution*. Le contexte correspond à toutes les informations (clic de souris, position de la souris sur un élément du diagramme, entrée du clavier, etc.) qui permet de déterminer quelle fonction doit être appelée. Par exemple, le cliquer-déplacer sur le bord d'un élément déclenche l'appel de la fonction *redimensionner*. La librairie facilite l'utilisation et l'écriture de ces fonctions, ce qui se révèle être très pratique. Bien qu'assez répétitive, l'écriture des fonctions peut être automatisée par des outils tels que Spray.

Les inconvénients de Graphiti sont donc d'une part les développements répétitifs et d'autre part la nécessité de connaitre la librairie offerte par Graphiti. Ces défauts sont cependant très vite oubliés devant sa prise en main rapide et son caractère intuitif. Graphiti permet en outre une très grande flexibilité en termes de personnalisation du comportement et du rendu visuel. Le concept de *fournisseur de fonctions* (*featureProvider*), permet de définir, selon un contexte déterminé (par exemple, le rôle de l'utilisateur vis-à-vis de l'outil), quelles sont les fonctionnalités possibles dans l'éditeur. Graphiti permet de librement alterner entre différents fournisseurs de services, et ce même durant l'exécution de l'éditeur. Cet aspect est fortement apprécié afin de ne pas avoir à dupliquer un éditeur pour les besoins du chef de projet et du développeur.

Comparatif des différents outils. Parmi les projets et outils étudiés, seuls le projet UML2 et l'outil Papyrus MDT permettent de concevoir des extensions du méta-modèle UML. UML2 propose un éditeur arborescent, tandis que Papyrus MDT fournit un éditeur graphique. Chaque éditeur possède ses forces et faiblesses. L'éditeur arborescent peut sembler compliqué à utiliser (notamment pour la création des associations) mais dispose de toutes les fonctionnalités pour

Caractéristiques / outil	GMF	Papyrus MDT	Graphiti
Approche			
Type d'approche	générative	générative	code
Adaptation de diagrammes UML	○	●	○
Personnalisation des éditeurs			
Diagrammes	◐	◐	●
Divers (e.g. panneau de propriétés)	○	◐	●
Facilité de personnalisation	○	◐	◐
Développement			
Itératif & Incrémental	◐	○	●
Assistants de création	◐	◐	○
Facilité de développement	○	○	●
Communauté & Développement			
Documentation (forum, wiki, etc.)	◐	◐	●
Évolution	○	◐	●

Table 40 – Conception d'éditeurs graphiques : les outils GMF, Papyrus MDT et Graphiti

concevoir une extension du méta-modèle (définition de méta-classes, sous-ensemble et redéfinition des propriétés, généralisation des concepts UML, etc.). Papyrus MDT fournit un éditeur graphique et le diagramme de classes UML peut servir à la définition des méta-modèles, mais il ne dispose d'aucun éditeur dédié à la conception de méta-modèles. Il manque donc de différentes fonctionnalités (e.g. conversion d'une classe en méta-classe) pour concevoir un méta-modèle. La visualisation graphique est cependant très utile et Papyrus MDT permet de simplifier certaines activités (telles que la création d'une association dans le méta-modèle) par rapport à UML2.

Pour la conception d'éditeurs graphiques, nous avons comparé les outils GMF, Papyrus MDT et Graphiti. Le comparatif de ces outils a fait l'objet de nombreuses études. Nous pouvons citer par exemple [Refsdal 2011] qui compare Graphiti et GMF. La table 40 illustre un comparatif des principales caractéristiques des différents outils. Le point le plus important concerne la personnalisation des éditeurs afin que ces derniers soient adaptables et dédiés à la méthode $\langle\mathsf{HOE}\rangle^2$. Ce point fait défaut aux deux éditeurs GMF et Papyrus MDT. En revanche, Papyrus MDT est bien mieux adapté pour la modélisation UML, et fournit déjà des éditeurs graphiques qu'il est possible d'étendre, ce qui en fait un bon candidat pour le développement d'extensions du méta-modèle. Cependant, son manque de flexibilité pour la personnalisation du comportement nous a dissuadés de l'utiliser. En ce qui concerne la communauté d'utilisateurs et l'évolution de l'outil, Papyrus MDT et Graphiti apportent beaucoup plus de réponses que GMF. En termes de développement néanmoins, Graphiti permet un développement beaucoup plus agréable que les deux autres outils basés sur du code GMF, bien que ne proposant aucun assistant permettant de réduire l'effort de développement (comme le tableau de bord de GMF ou l'assistant de création de menu de propriétés de Papyrus MDT). L'étude réalisée par [Refsdal 2011] indique en effet que GMF est moins apprécié que Graphiti [6]. En outre, la phase d'apprentissage de Graphiti

6. Étude réalisée sur douze étudiants en Technologie de l'Information, parmi lesquelles sept personnes préfèrent Graphiti, soit un total de 58% des étudiants.

est vite compensée par le temps de développement plus court que pour GMF, toujours selon l'étude réalisée par [Refsdal 2011] [7].

Nous avons chronologiquement testé GMF, puis Papyrus MDT, et avons été freinés à deux reprises par la complexité du code généré, le nombre de bogues rencontrés, le peu de documentation et le manque de support. En raison de la complexité du développement engendrée, nous avons privilégié l'usage de Graphiti pour concevoir les éditeurs graphiques dédiés au langage $\langle \mathsf{HOE} \rangle^2$. La partie suivante illustre les développements effectués en termes d'éditeurs graphiques.

2.3 Choix de la solution et illustration

Au vu de l'étude effectuée sur les outils, nous avons choisi d'utiliser UML2 et Papyrus MDT conjointement pour la définition des méta-modèles. Pour la conception des éditeurs graphiques, nous avons privilégié Graphiti, dont l'approche par codage est plus longue, mais permet une personnalisation totale des éditeurs.

Définition des méta-modèles $\langle \mathsf{HOE} \rangle^2$ avec UML2 et Papyrus MDT. Les différents méta-modèles du langage $\langle \mathsf{HOE} \rangle^2$ présentés au chapitre 6, aussi bien pour la partie *produit* que pour la partie *gestion de projet* ont été réalisés à l'aide de l'éditeur arborescent présenté dans la figure 120. La figure 120 illustre le méta-modèle de l'analyse du besoin $\langle \mathsf{HOE} \rangle^2$ saisi dans l'éditeur arborescent du projet UML2. Deux stéréotypes sont utilisés, « Metamodel, EPackage » pour indiquer que le modèle conçu est un méta-modèle étendant le méta-modèle UML. Dans cette figure, nous pouvons voir l'usage des stéréotypes « Metaclass » afin de spécifier les concepts de notre méta-modèle. Pour chacun, nous avons étendu les concepts UML par l'usage de la généralisation. Par exemple, le concept d'*acteur* dans le méta-modèle d'analyse du besoin spécialise la méta-classe UML *Actor*. De nouvelles propriétés viennent s'ajouter ou redéfinir les propriétés existantes. Il est également possible de définir de nouvelles associations dans le méta-modèle, en spécialisant les relations déjà existantes.

Conjointement à l'éditeur arborescent du projet UML2 (cf. fig. 120), nous avons utilisé les diagrammes fournis par Papyrus MDT afin d'avoir une représentation graphique des méta-modèles créés (cf. fig. 121). Dans la figure 121, nous pouvons distinguer les éléments nouveaux du méta-modèle d'analyse du besoin $\langle \mathsf{HOE} \rangle^2$ des éléments du méta-modèle UML, dont la mise en forme est différente et pour lesquels les noms sont préfixés par celui du méta-modèle de provenance. La représentation graphique est beaucoup plus lisible et manipulable, notamment pour la création des associations du méta-modèle.

Une fois les méta-modèles réalisés, l'usage de l'outil EMF nous a permis de générer les classes et interfaces Java des concepts, afin qu'ils soient manipulables dans tous les autres outils de modélisation de l'environnement Eclipse MDT.

Développement des éditeurs graphiques avec Graphiti. Les éditeurs graphiques $\langle \mathsf{HOE} \rangle^2$ ont été construits sur la base de Graphiti. Nous avons privilégié un multi-éditeur par phase. Chaque multi-éditeur est constitué de l'ensemble des diagrammes de la phase en question. Un greffon a été développé par multi-éditeur et un greffon par diagramme. Ainsi, pour la phase d'analyse du besoin, un greffon a été développé pour l'éditeur de la phase et deux greffons pour

7. Trois à quatre semaines au total pour GMF dont 100% du temps est consacré au développement ; deux semaines et demi pour Graphiti dont deux semaines sont consacrées à l'apprentissage, la moitié d'une semaine au développement.

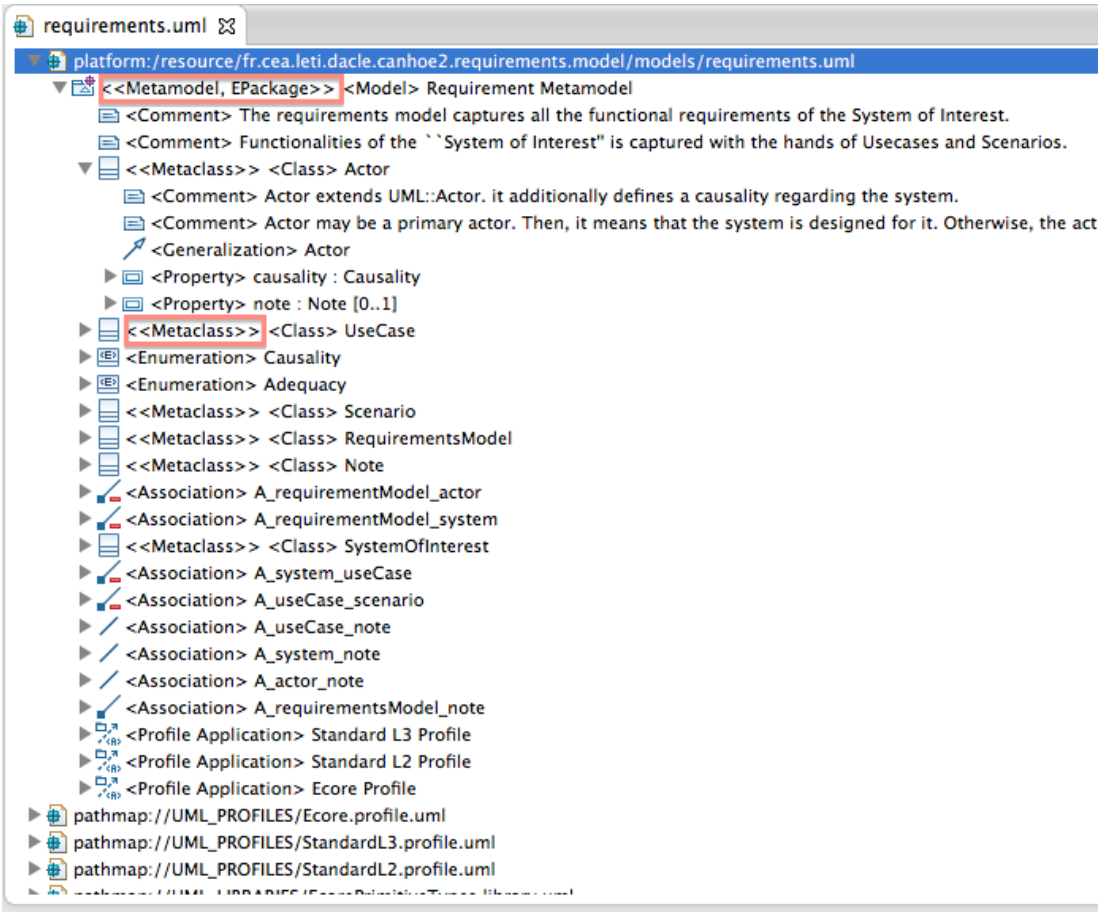


FIGURE 120 – Utilisation de l'éditeur arborescent du projet UML2 pour la conception du méta-modèle d'analyse du besoin ⟨HOE⟩[2]

les diagrammes d'analyse du besoin et de scénarios. Pour chaque diagramme, un ensemble de fonctionnalités basiques (création, ajout, suppression, déplacement, redimensionnement, etc.) et avancées (clic et double-clic, édition directe de texte, menu déroulant, etc.) est fourni pour chaque concept du méta-modèle illustré dans le diagramme. Diverses fonctionnalités de l'éditeur graphique ont été développées, telles que des menus de propriétés pour chaque concept, des fonctionnalités automatiques (ajout, redimensionnement et placement) ont été ajoutées.

La figure 122 illustre le résultat du développement de l'éditeur de la phase d'analyse du besoin. L'éditeur est divisé en onglets, le premier onglet permet d'afficher le diagramme d'analyse du besoin pour modéliser le système, ses acteurs et ses cas d'utilisation. Les autres onglets permettent de modéliser les scénarios dans des diagrammes de scénarios. Le panneau de propriétés

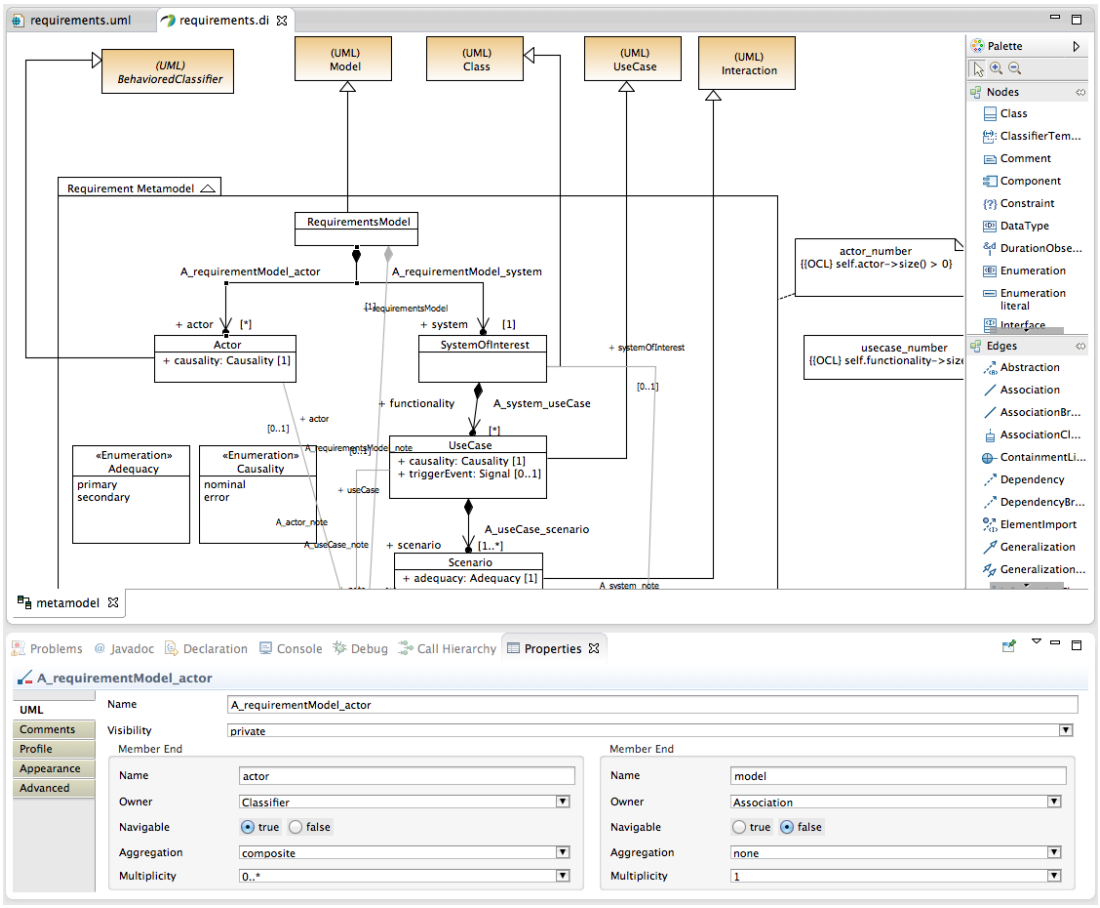


FIGURE 121 – Utilisation du diagramme de classes de Papyrus MDT pour la conception du méta-modèle d'analyse du besoin $\langle \mathsf{HOE} \rangle^2$

est dédié à chaque concept du méta-modèle.

EN RÉSUMÉ

- ✓ *Étude des projets UML2 et GMF, des outils Papyrus MDT et Graphiti pour la définition de méta-modèles UML et la conception des éditeurs graphiques du langage* $\langle \mathsf{HOE} \rangle^2$
- ✓ *Définition des méta-modèles avec UML2 et Papyrus MDT utilisés conjointement*
- ✓ *Conception des éditeurs graphiques avec Graphiti pour ses nombreuses fonctionnalités et qualités (approche, personnalisation, éovlution.)*

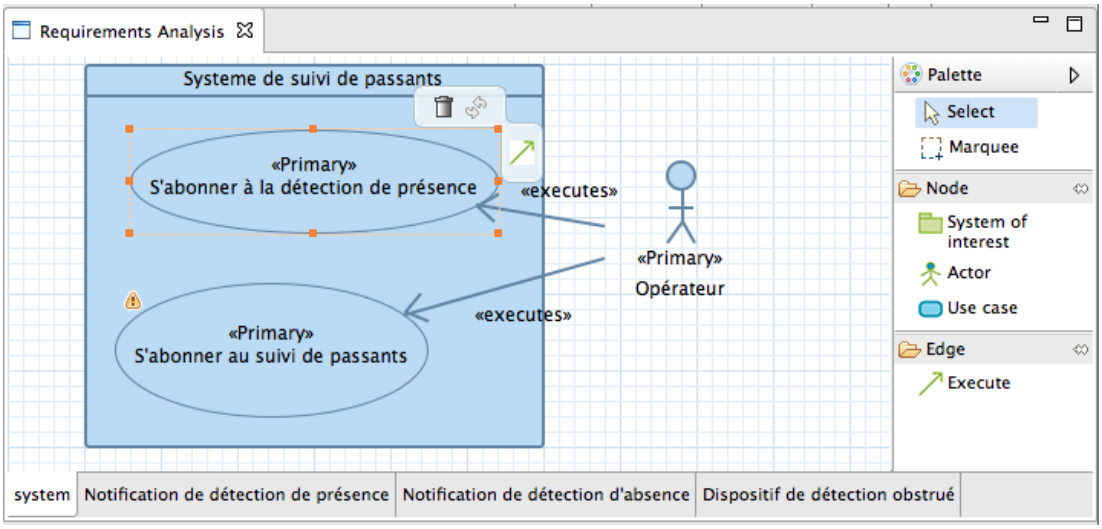


FIGURE 122 – Éditeur graphique pour la phase d'analyse du besoin

3 Support au processus : gestion des rôles

Le support au processus est nécessaire pour assurer le guidage aux travers des quatre phases du processus ⟨HOE⟩^{2}. De plus, l'outil doit être multi-utilisateurs et notamment offrir différentes vues pour le développeur et le chef de projet. Dans cette section, nous montrons comment le support au processus a été considéré dans le cadre du développement de l'outil CanHOE2. Cette section n'aborde que l'aspect gestion des rôles du processus qui a été implémenté.

3.1 Rappel de nos besoins

La gestion des rôles doit être pensé en premier lieu au niveau de l'outil dans sa globalité et en second lieu au niveau des éditeurs graphiques de l'outil CanHOE2. Nous avons vu lors de la présentation de la plate-forme Eclipse que celle-ci gère nativement les aspects multi-utilisateurs et plusieurs rôles, notamment au moyen de la définition des perspectives. Cette gestion n'est en revanche que rarement possible dans les éditeurs graphiques. Pour cela, le comportement de l'éditeur doit être adapté vis-à-vis de l'utilisateur qui l'utilise. Par exemple, dans le cas des diagrammes ⟨HOE⟩^{2}, les actions sur les modèles possibles du chef de projet se résument aux activités d'initialisation et de clôture des phases. Pour le développeur, toutes les activités de modélisation, telles que la création de nouveaux éléments depuis une palette d'outils, ou encore l'édition ou la suppression d'éléments existants doivent être réalisables.

Nous avons déjà fait le choix dans la section précédente d'utiliser l'outil Graphiti pour concevoir les éditeurs graphiques. L'étude sur les outils ne traitera donc que de cet outil et des différentes solutions d'implémentation de la gestion de rôles.

3.2 Études des solutions et des outils existants

Graphiti n'intègre pas nativement la gestion de plusieurs rôles afin d'adapter le comportement des éditeurs. Nous avons envisagé durant cette thèse deux solutions possibles.

Conception d'un éditeur pour chaque rôle. La première implémentation consiste simplement à réaliser pour chaque phase du processus et pour chaque rôle du processus un éditeur et des diagrammes distincts. Cette approche nous permettrait d'obtenir un comportement des éditeurs dédié au rôle de l'utilisateur qui l'utilise. Cette solution est viable dans le sens où le rôle de l'utilisateur ne varie pas durant l'ouverture des diagrammes. Elle nécessite néanmoins beaucoup de développements redondants entre les deux outils.

Le fournisseur de fonctionnalités de Graphiti. Si Graphiti n'intègre pas la gestion de rôles, il est toutefois possible de l'obtenir au travers du concept de *fournisseur de fonctionnalités*[8]. Un fournisseur de fonctionnalités dans Graphiti est simplement une classe permettant de définir, en fonction d'interactions de l'utilisateurs (un cliquer-déplacer depuis la palette à outils, un clic sur le nom d'un élément affichée dans le diagramme ou sur une bordure de l'élément, etc.) quelle fonctionnalité basique ou avancée il est possible de déclencher (création et affichage d'un nouvel élément, édition directe de texte, redimensionnement, etc.). L'intérêt du fournisseur de fonctionnalités Graphiti est de pouvoir être changé à tout moment, même en cours d'utilisation. Il est ainsi possible de définir un fournisseur de fonctionnalités pour chaque rôle de l'outil sans avoir à dupliquer l'éditeur complet. Cela permet une forte réutilisation, notamment des fonctionnalités communes aux deux rôles (par exemple, les deux rôles peuvent modifier la représentation graphique – qui leur est personnelle – mais seul le développeur peut modifier le modèle sous-jacent).

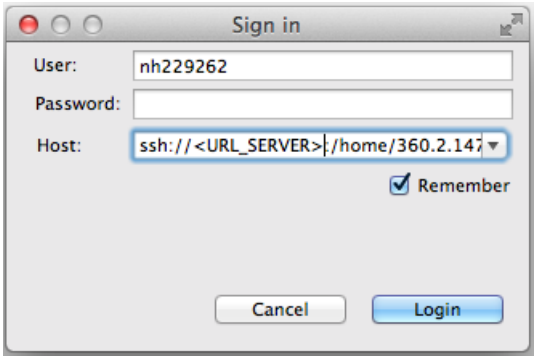


FIGURE 123 – Fenêtre d'authentification sur le serveur

3.3 Choix de la solution et illustration

Nous avons donc choisi d'utiliser les perspectives Eclipse pour la gestion des rôles globalement dans CanHOE2. Pour les éditeurs graphiques, nous avons privilégié l'usage d'un *fournisseur de fonctionnalités* pour chaque rôle plutôt que de dupliquer les éditeurs. Cette solution

8. En anglais, *featureProvider*.

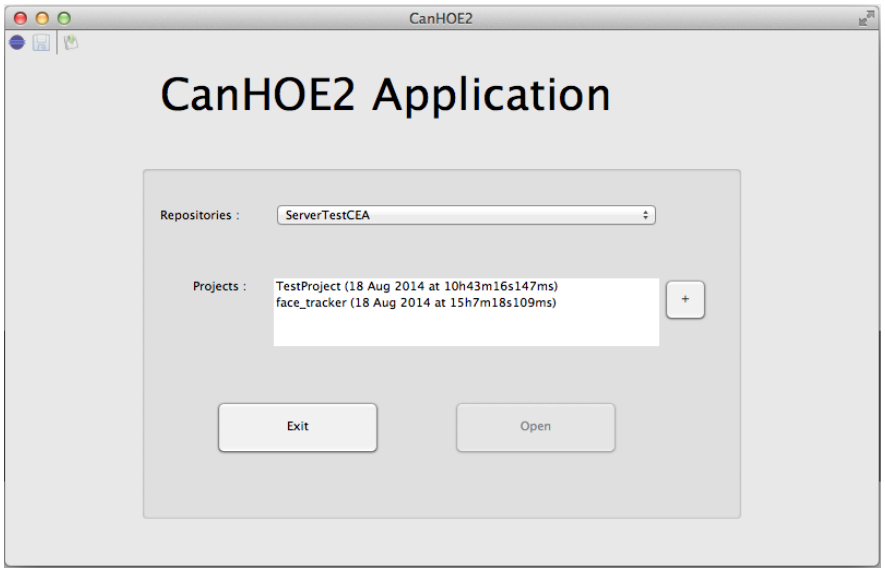


FIGURE 124 – Écran d'accueil et sélection des projets

nous permet plus de flexibilité et de réutilisation, tout en réduisant le développement. Dans cette partie, nous présentons les deux perspectives correspondant aux deux rôles du processus.

Authentification et identification des rôles. L'identification du rôle de l'utilisateur s'effectue au moment de son authentification. Lors de l'ouverture de CanHOE2, une première fenêtre permet de s'authentifier et se connecter à un serveur d'authentification particulier (cf. fig. 123). L'authentification se fait au moyen du protocole LDAP [9].

Une fois authentifié, un écran d'accueil (cf. fig. 124) permet au participant de sélectionner un projet parmi la liste des projets dans lesquels il est impliqué, que ce soit en tant que chef

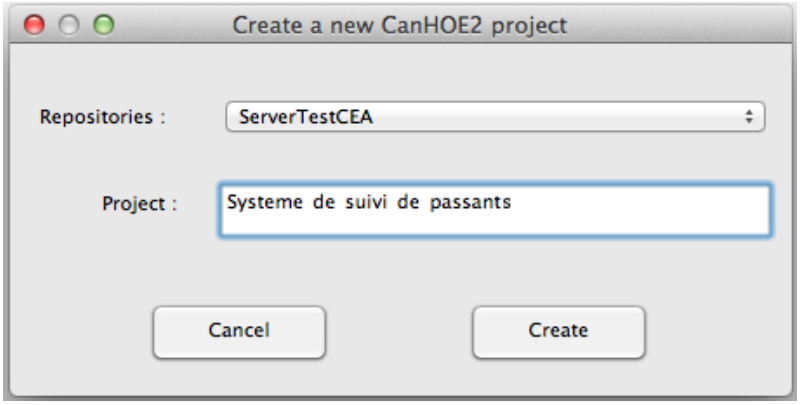


FIGURE 125 – Création d'un projet

9. En anglais, LDAP.

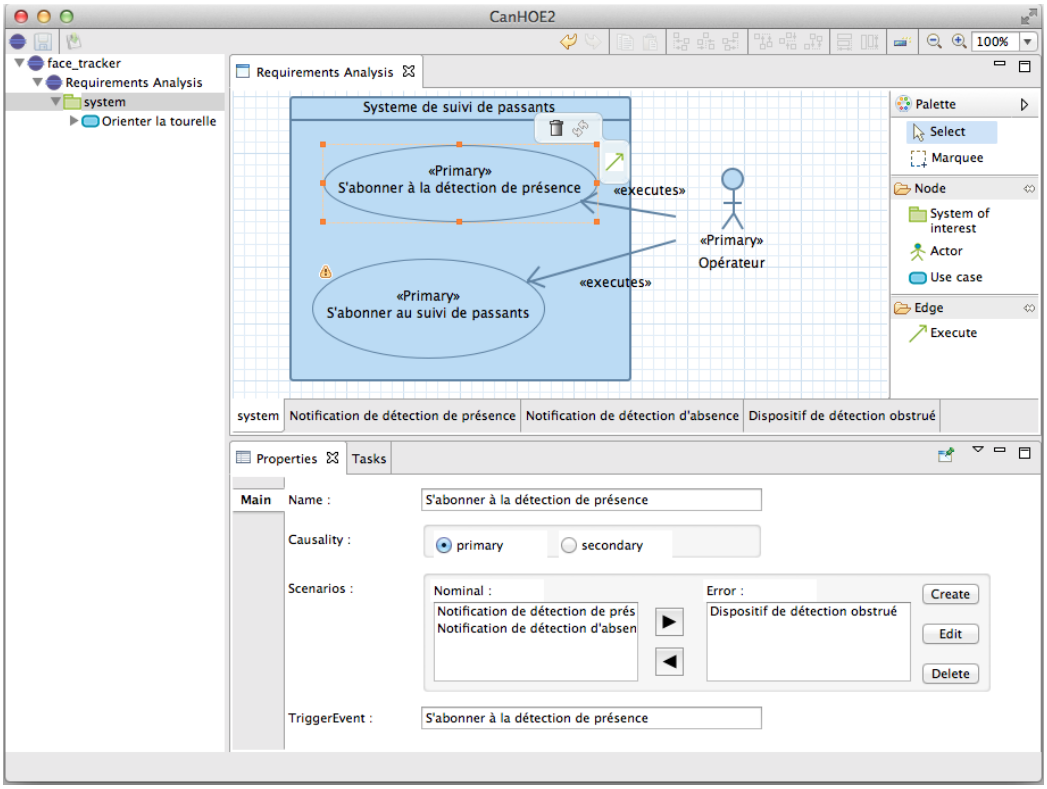


FIGURE 126 – Perspective *développeur*

de projet ou en tant que développeur. Il a également la possibilité d'en créer un nouveau (cf. fig. 125). Lorsque le projet est sélectionné, l'écran d'accueil est remplacé par une perspective propre au rôle du participant vis-à-vis du projet sélectionné. Les figures 126 et 127 illustrent les deux perspectives, respectivement celle du chef de projet et du développeur.

Perspective « développeur ». La perspective *développeur* (cf. fig. 126) permet au développeur de réaliser les tâches de modélisation assignées par le chef de projet. Elle est principalement constituée d'une zone d'édition, dans laquelle le développeur peut ouvrir le modèle. Une palette et un panneau de propriétés accompagnent la zone d'édition pour éditer le modèle. Un explorateur de modèles à gauche permet de naviguer dans le modèle et ouvrir l'éditeur correspondant à la phase.

Perspective « chef de projet ». La perspective du chef de projet est illustrée par la figure 127. Cette perspective est structurée en plusieurs parties. À gauche, le même *explorateur de modèles* que le développeur permet de naviguer dans l'ensemble des modèles produits durant les quatre phases de la méthode $\langle \mathsf{HOE} \rangle^2$. La zone centrale utilise les mêmes éditeurs que le développeur, mais seulement en mode visualisation et non édition. Elle ne possède donc pas de palette d'outils et les seules actions pouvant être réalisées par le chef de projet sont l'initialisation et la clôture des phases, ainsi que la modification visuelle du diagramme (sans toutefois modifier

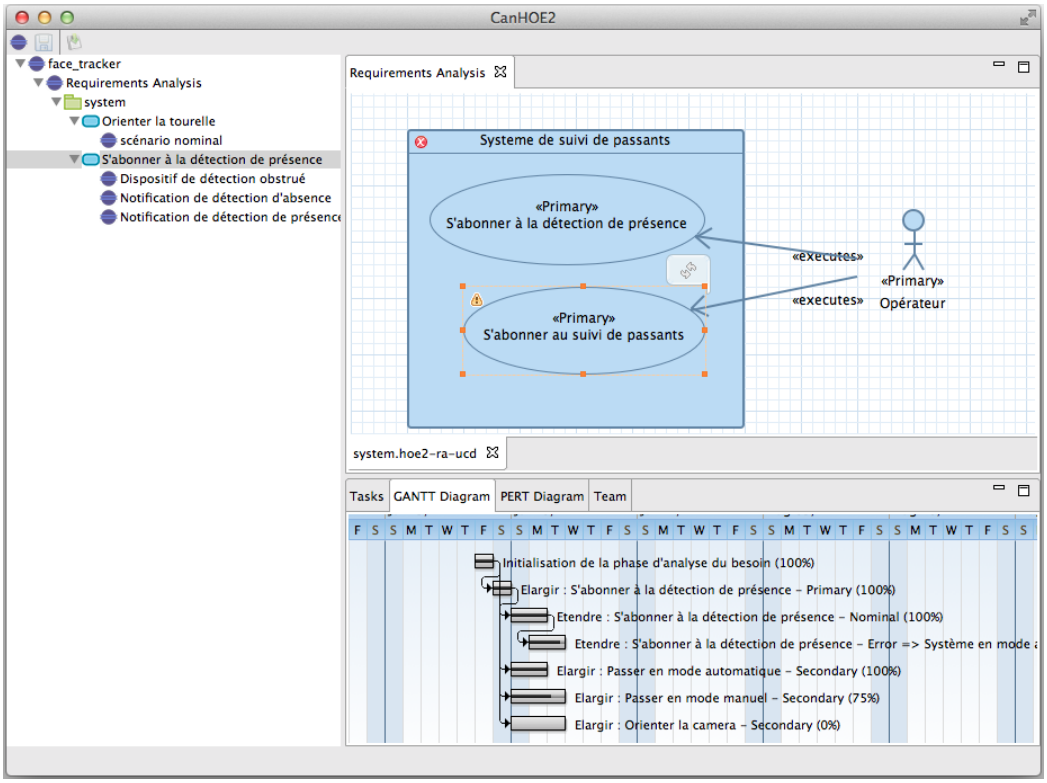


FIGURE 127 – Perspective *chef du projet*

le modèle) lui permettant d'organiser graphiquement les éléments du modèle selon son désir.

Les deux figures 126 et 127 illustrent l'ouverture du même éditeur, mais avec un *fournisseur de fonctionnalités* différents. Dans la figure 127, les fonctionnalités fournies permettent par exemple la création de la palette, la suppression d'un élément (représenté par l'icône au dessus du premier cas d'utilisation) et l'assistant de connexion (représenté par l'icône). L'icône de rechargement () est une fonctionnalité commune aux deux rôles. Dans le cas de la figure 127, il n'y a ni palette, ni possibilité de modifier le modèle.

EN RÉSUMÉ

- ✓ *Implémentation d'une perspective neutre quand le rôle de l'utilisateur est indéfini*
- ✓ *Implémentation de deux perspectives dédiées pour le chef de projet et le développeur et de deux comportements distincts des éditeurs graphiques*
- ✓ *Authentification de l'utilisation et identification de son rôle au moyen du protocole LDAP*

4 Support à la gestion de projet

Cette section aborde le support par l'outillage de la gestion de projet présentée dans le chapitre 8.

4.1 Rappel de nos besoins

L'outil CanHOE2 dédié nécéssite d'incorporer une gestion de projet intégrée au sein de l'outil. Cette gestion de projet se limite pour le moment à l'organisation des équipes et à la planification des tâches. Pour implémenter cette gestion de projet, il est nécessaire en premier lieu d'implémenter le méta-modèle de gestion de projet au sein de l'outil. Une fois le méta-modèle implémenté, nous devons développer des vues spécifiques au chef de projet pour l'organisation de son équipe, la création, planification et assignation des tâches. Le développeur nécessite également des vues pour la réception des tâches. Certaines vues spécifiques au chef de projet doivent inclure des diagrammes courants de gestion de projet tels que les diagrammes de Gantt par exemple. D'autres vues nécessitent de représenter des données tabulaires Ces vues spécifiques nécessitent d'étudier les solutions existantes permettant de les implémenter.

4.2 Études des solutions et des outils existants

Nous avons étudié dans les parties précédentes les vues permettant la définition d'extensions du méta-modèle UML. Nous présentons dans cette partie deux projets sur lesquels il est possible de s'appuyer pour implémenter la gestion de projet.

WindowBuilder. Eclipse inclut nativement un ensemble de composants de la librairie graphique SWT [10]. Avec SWT, il est possible de concevoir des boutons, champs de texte, tableaux, etc. et de capturer différentes actions de l'utilisateur (souris et claviers). La création de vues à base de composants SWT dans Eclipse est simplifiée par l'utilisation de l'éditeur WindowBuilder [11]. Cet éditeur composé de deux onglets, *source* et *design* permet d'assembler graphiquement des composants SWT dans une vue Eclipse. Dans l'onglet *source*, il est possible de visualiser et de modifier le code source de la vue créée et définir les méthodes de remplissage des composants SWT (les tables par exemple) . Les deux onglets sont synchronisés, afin que chaque changement dans l'un des deux onglets *source* ou *design* se répercutent sur le second. L'outil est intuitif et la création de nouvelles vues s'effectue très rapidement.

Le projet Nebula. Nebula [12] est un projet Eclipse réunissant dans une même librairie un ensemble de composants SWT avancés. Les composants ne sont pas spécifiques à la gestion de projet (la librairie propose par exemple des composants pour simuler un oscilloscope, ou une sonde de température), mais elle contient quelques éléments intéressants, tels qu'un diagramme de Gantt. La librairie est facile à utiliser et plutôt bien documentée, les composants peuvent être personnalisés et être intégrés dans des vues ou dans d'autres composants.

10. En anglais, Standard Widget Toolkit (SWT).
11. `http://www.eclipse.org/windowbuilder/`
12. `http://www.eclipse.org/nebula/`

Le projet BIRT. Business Intelligent and Reporting Tools (BIRT) [13] est un outil permettant la réalisation de rapport intelligents. Ces rapports permettent de construire un ensemble important de graphiques standards (e.g. camemberts, nuages de points) et avancés (e.g. diagramme de Gantt). Les rapports peuvent être intégrés dans des vues et les données peuvent être importées depuis une base de données, ou un schéma XML. Il est également possible, à partir de greffons spécifiques, d'importer des données depuis un modèle EMF (et donc UML) ou un annuaire LDAP. La syntaxe est très complète et permet de créer des vues de qualité. En outre, BIRT est très bien documenté et en constante évolution.

4.3 Choix de la solution et illustration

Nous n'avons pas été en mesure par manque de temps de tester tous les outils. Notamment, BIRT semble correspondre à nos besoins pour la création de vues spécifiques de gestion de projet, mais nécessite un temps d'apprentissage plus important, en particulier pour prendre en main la synchronisation entre des rapports BIRT avec des données provenant d'un modèle EMF ou d'un annuaire LDAP. Nous avons donc privilégié les composants disponibles nativement dans SWT, tels que les tables, pour les vues dédiées à l'organisation des équipes et la planification des tâches, ainsi que les composants avancés du projet Nebula pour le diagramme de Gantt.

Pour cela, nous avons conçu onze greffons pour la gestion de projet : deux greffons internes *canhoe2.projectManagement* et *canhoe2.projectManagement.task* pour l'intégration de la gestion de projet avec les autres greffons de l'outil CanHOE2, six vues présentant différentes facettes de la gestion de projet au sein de l'outil, et trois greffons *canhoe2.projectmanagement.activity.model*, *canhoe2.projectmanagement.participant.model* et *canhoe2.projectmanagement.project.model*, contenant la définition du méta-modèle de gestion de projet présenté dans le chapitre 8 et le code généré avec EMF de ces modèles sous forme de classes et d'interfaces pour chaque concept.

Définition du méta-modèle de gestion de projet. Afin d'outiller la gestion de projet au sein de CanHOE2, nous avons commencé par définir le méta-modèle de gestion de projet présenté dans le chapitre 8 au sein de l'outil. Tout comme pour les méta-modèles des quatre phases du processus, nous avons utilisé conjointement les outils UML2 et Papyrus MDT.

L'implémentation du méta-modèle réalisée, il est désormais possible d'outiller la gestion de projet. Nous nous sommes limités aux aspects étudiés dans le chapitre 8 : la gestion des équipes et l'organisation des activités.

Organisation des équipes. La gestion des ressources (physiques et humaines) est l'une des activités fondamentales de la gestion de projet. Nous nous sommes focalisés sur la gestion des équipes, avec le concept *Participant* dans le méta-modèle. Lorsqu'un utilisateur crée un projet, il en devient le chef de projet. Une instance du concept *Participant* est alors créée et associée au projet en tant que chef de projet. Aucun développeur n'est encore présent. Afin de le permettre, nous avons ajouté une vue dédiée à la gestion des équipes dans la perspective *chef de projet* (cf. fig. 128). Cette vue a été implémentée dans le greffon *canhoe2.view.teamProjectManagerView*. Une fois connecté à un projet dans lequel l'utilisateur de l'outil est chef de projet, ce dernier peut accéder à cette vue depuis sa perspective.

13. `http://www.eclipse.org/birt/`

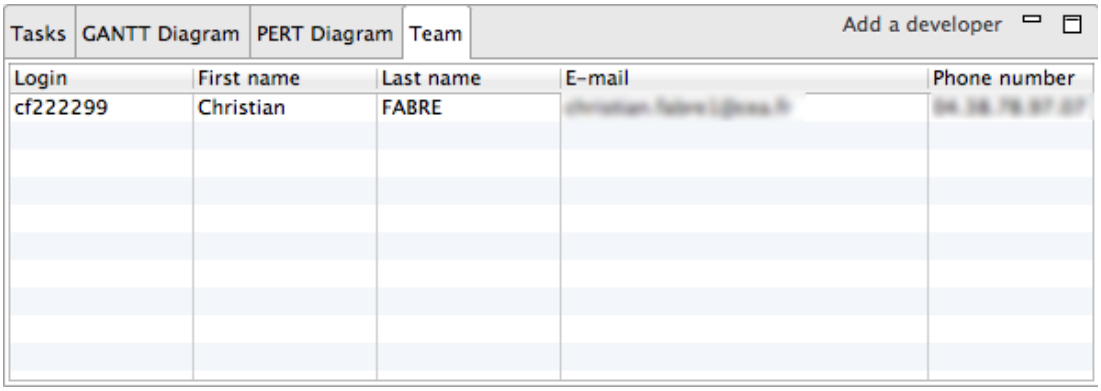


FIGURE 128 – Tableau de bord des participants

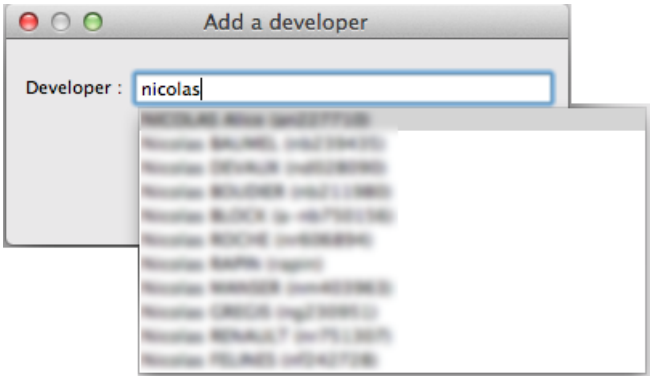
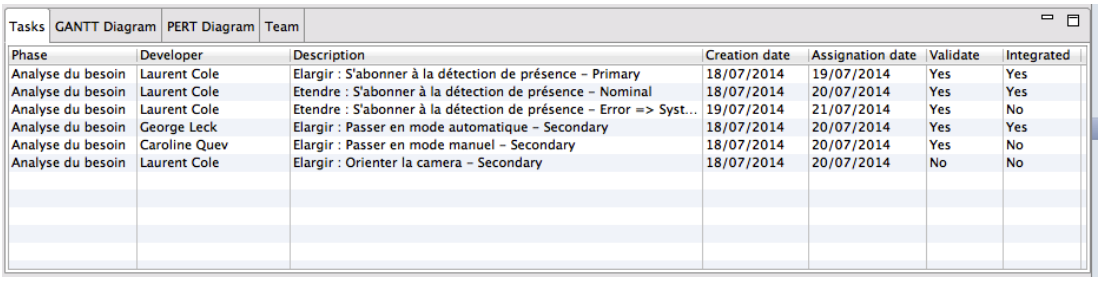


FIGURE 129 – Ajout d'un développeur au projet

Cette vue se présente comme un tableau de bord des participants à un projet. Elle contient la liste des développeurs attachés au projet, ainsi que les différentes données les définissant. Dans cette vue, le chef de projet peut décider d'ajouter un nouveau participant au projet. L'ajout s'effectue par l'appel d'un assistant. Cet assistant a été développé dans un greffon indépendant *canhoe2.views.addDeveloperView*. La figure 129 illustre cet assistant. Il contient un unique champ de recherche dans lequel le chef de projet peut saisir une partie du nom ou du prénom du développeur. La recherche s'effectue auprès d'un annuaire LDAP. Une fois le développeur ajouté au projet, le chef de projet peut lui assigner une ou plusieurs tâches à réaliser.

Planification des tâches. L'outil CanHOE2 permet la gestion des feuilles de tâche et de leur cycle de vie (INITIALISÉE, ASSIGNÉE, RÉALISÉE, VALIDÉE et INTÉGRÉE ou REJETÉE). Le développement actuellement réalisé ne permet de supporter que les deux premiers états de la feuille de tâche, à savoir son initialisation et son assignation à un développeur. Nous avons pour cela commencé à développer le greffon *canhoe2.projectManagement.task* pour gérer les différentes transactions entre le chef de projet et le développeur, à savoir l'assignation de la tâche par le chef de projet et la réception de cette dernière par le développeur. Deux vues spécifiques embarquées

dans les greffons *canhoe2.views.taskDeveloperView* et *canhoe2.views.taskProjectManagerView* ont été réalisées.



Phase	Developer	Description	Creation date	Assignation date	Validate	Integrated
Analyse du besoin	Laurent Cole	Elargir : S'abonner à la détection de présence - Primary	18/07/2014	19/07/2014	Yes	Yes
Analyse du besoin	Laurent Cole	Etendre : S'abonner à la détection de présence - Nominal	18/07/2014	20/07/2014	Yes	Yes
Analyse du besoin	Laurent Cole	Etendre : S'abonner à la détection de présence - Error => Syst...	19/07/2014	21/07/2014	Yes	No
Analyse du besoin	George Leck	Elargir : Passer en mode automatique - Secondary	18/07/2014	20/07/2014	Yes	Yes
Analyse du besoin	Caroline Quev	Elargir : Passer en mode manuel - Secondary	18/07/2014	20/07/2014	Yes	No
Analyse du besoin	Laurent Cole	Elargir : Orienter la camera - Secondary	18/07/2014	20/07/2014	No	No

FIGURE 130 – Tableau de bord des tâches

La figure 130 illustre une vue du chef de projet définissant la liste des tâches assignées aux développeurs. Différentes informations sont disponibles pour définir et organiser les tâches : la phase à laquelle la tâche s'applique, le développeur à qui la tâche est assignée, les dates de création et d'assignation, et enfin deux informations permettant de définir si la tâche est validée ou intégrée.

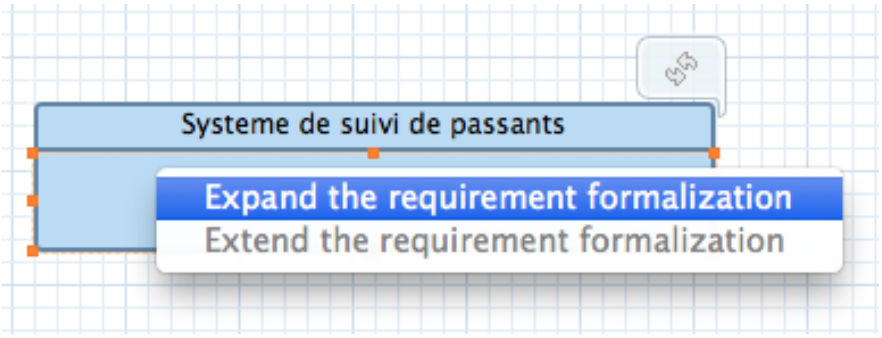


FIGURE 131 – Création des tâches

L'assignation des tâches se fait par le chef de projet depuis les modèles ouverts. La figure 131 illustre la proposition de création de tâche sur le modèle d'analyse du besoin ouvert dans l'éditeur Graphiti. La liste des propositions est gérée par le greffon *canhoe2.projectManagement.task*. Ici, les deux tâches de modélisation de l'activité de formalisation de besoin sont proposées. La première permet de créer une tâche pour ajouter de nouveaux acteurs et cas d'utilisation au système. Elle est active, puisque le modèle d'analyse du besoin a été initialisé. Le seconde permet d'ajouter des scénarios à un cas d'utilisation. Elle est pour le moment inactive puisqu'il n'y a encore aucun cas d'utilisation modélisé pour le système.

Les figures 132 et 133 illustrent un assistant permettant la création et l'assignation d'une tâche. Elle permet de décrire la tâche et de choisir parmi la liste des développeurs impliqués dans le projet celui à qui la tâche sera assignée. Le contenu du premier champ permettant de sélectionner le modèle partiel qui sera fourni au développeur est alimenté par le greffon *canhoe2.projectManagement.task*. Ce greffon implémente une classe Java par tâche et permet, à partir de l'élément sélectionné sur le diagramme, de définir les éléments qui constitueront le

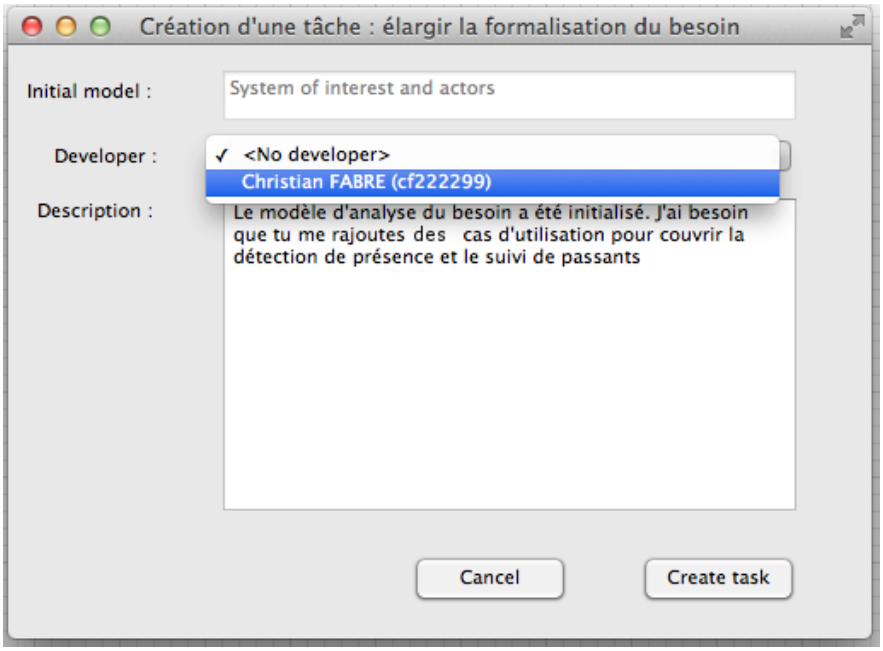


FIGURE 132 – Assignation d'une feuille de tâche d'élargissement de la formalisation du besoin

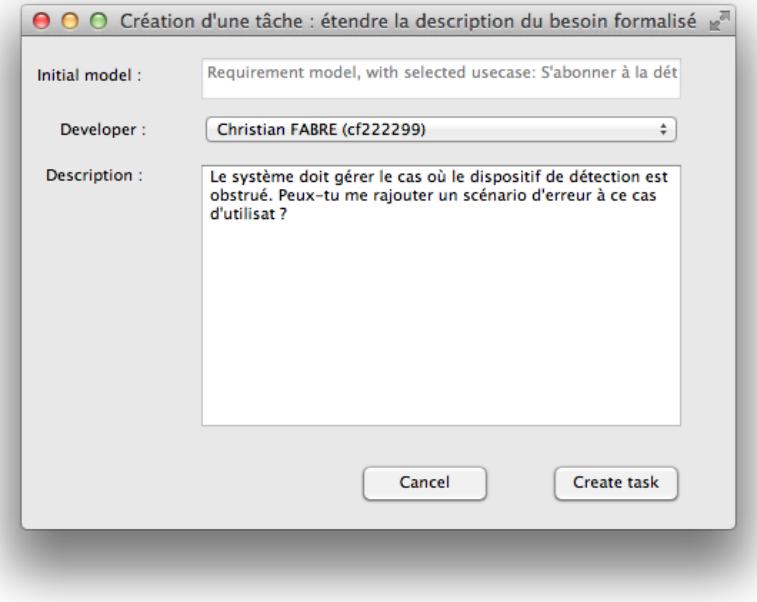


FIGURE 133 – Assignation d'une tâche d'extension de la description du besoin formalisé

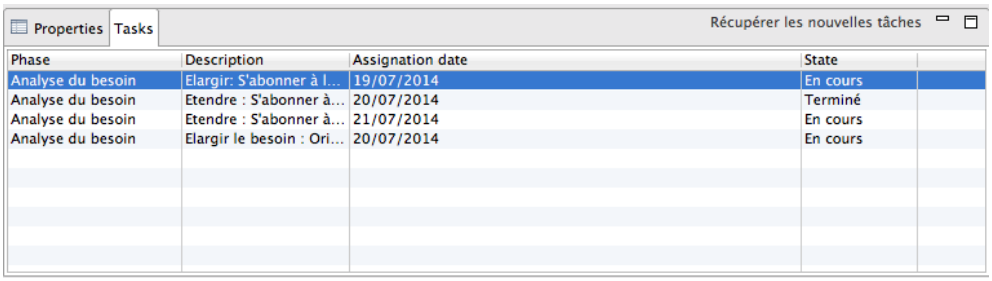


FIGURE 134 – Réception d'une feuille de tâche

modèle fourni au développeur. Dans le cas de la figure 133, si un cas d'utilisation a été sélectionné, cette classe fournit le modèle composé de l'unique cas d'utilisation et de ses scénarios, ainsi que de l'ensemble des acteurs pouvant l'exécuter. Au terme de cette vue une feuille de tâche est dans l'état ASSIGNÉE. Elle est envoyée au développeur qui doit alors la réaliser.

La feuille de tâche dans l'état ASSIGNÉE peut être réceptionnée par le développeur. Le greffon *canhoe2.views.taskDeveloperView* permet de contribuer à la perspective du développeur en y ajoutant une vue de réception des tâches. Cette vue est illustrée par la figure 134. Elle permet au développeur de visualiser l'ensemble des tâches qui lui sont assignées et de récupérer les nouvelles en cliquant sur le bouton situé en haut à droite de la vue. Il peut également double-cliquer sur une tâche afin d'ouvrir le modèle associé dans la zone d'édition et réaliser l'activité de modélisation correspondante.

Une fois la tâche réalisée par le développeur, ce dernier doit la retourner au chef de projet, pour que ce dernier puisse la valider et l'intégrer, ou au contraire la refuser. Ces aspects nécessitent des vues et outils qui sont en cours de développement.

Planification du développement et diagramme de Gantt. Afin de permettre au chef de projet de suivre le développement et planifier les différentes activités, nous avons commencé à mettre en place des diagrammes traditionnels de gestion de projet. Le diagramme de Gantt y est représenté, avec toutefois des différences dans son utilisation. La figure 135 illustre l'usage du diagramme de Gantt. Sur l'axe horizontal, nous conservons le temps du projet. Sur l'axe vertical, nous pouvons voir la liste des tâches, leurs durées et états d'avancement.

Nous avons identifié plusieurs évolutions envisageables dans cette vue. Par exemple, il serait intéressant de regrouper l'ensemble des tâches par phase et de permettre au chef de projet de ne visualiser qu'une phase en question. Cette vue, au même titre que les autres vues de gestion de projet, manipulent des objets *Tâche* et *Feuille de tâche*, contenant un ensemble d'information qu'il est possible d'exploiter.

EN RÉSUMÉ

- ✓ *Définition du méta-modèle de gestion de projet dans CanHOE2*
- ✓ *Intégration de la gestion de projet en phase préliminaire uniquement*
- ✓ *Implémentation de différentes vues pour la gestion des équipes et des tâches*

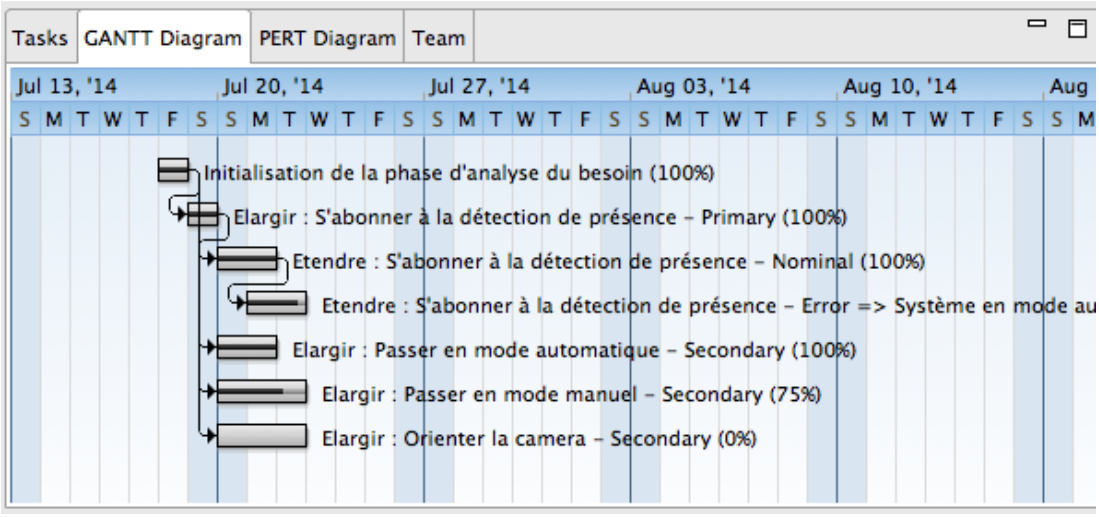


FIGURE 135 – Diagramme de Gantt dans CanHOE2

5 Support à la gestion de versions

L'outillage CanHOE2 doit également favoriser l'établissement d'une organisation efficace du développement et des artefacts produits. Nous nous sommes pour cela intéressés à l'organisation des développements dans les dépôts de révision de code, ainsi que la génération de documentation et de code. La génération d'artefacts est bien couverte par les outils existants, tandis que le premier point l'est beaucoup moins. En effet, l'organisation du code du système en cours de développement dans des dépôts est rarement gérée et laissée à des outils tiers. Nous avons jugé utile de nous concentrer sur ce point.

5.1 Rappel de nos besoins

Afin de permettre une organisation efficace du code, il est nécessaire de mettre en place un dépôt de modèles partagé par l'ensemble des participants à un projet. La gestion de versions nécessite d'être pleinement intégrée dans CanHOE2 afin de pouvoir l'utiliser en toute transparence. L'utilisation transparente permet d'éviter d'alourdir les différents participants à un projet avec des tâches non relatives au projet lui-même. L'usage d'outils de gestion de versions peut se révéler fructueux si tous les participants d'un projet les utilisent convenablement ou au contraire peut sensiblement affecter le développement dans le cas d'une utilisation maladroite. L'intégration transparente d'un outil de gestion de versions permet donc de limiter le risque d'utilisation tout en profitant de ses bénéfices pour la gestion des versions des modèles.

5.2 Études des solutions et des outils existants

Il existe de nombreux outils de gestion de versions, tels que Subversion, Git et Mercurial. Chaque outil possède son organisation et ses caractéristiques propres. Les trois outils se distinguent tout d'abord sur leur méthode de synchronisation. Dans Subversion, il existe un dépôt centralisé principal et plusieurs dépôts de travail. Dans Git et Mercurial, il n'existe pas de dépôt

principal, et le fonctionnement courant est décentralisé. Les trois outils diffèrent également sur d'autres points, tels que la définition d'un *commit* [14], ou la sémantique donnée aux concepts de branche ou de tag. Par exemple, Subversion organise le développement dans différents dossiers : *tronc*, *tag* et *branche*. Il ne fournit pas de commande spécifique pour la création d'une branche, ou d'un tag, ni même de sémantique sur ces concepts, mais reposent sur une convention d'organisation du dépôt dans différents dossiers et s'appuie sur des commandes classiques de copie et de fusion. À l'inverse, Git et Mercurial ne proposent aucune organisation de l'arborescence, mais s'appuient sur des mécanismes internes pour définir les branches et les tags.

5.3 Choix de la solution et illustration

Nous avons privilégié l'utilisation de Git, offrant de nombreuses fonctionnalités nécessaires pour l'outil CanHOE2. L'intégration transparente de Git au sein de CanHOE2 a nécessité le développement d'un greffon interne *canhoe2.git*. Ce greffon prend en charge la plupart des commandes natives Git, telles que la création de branches, les commits, la synchronisation à un serveur, etc. Il exporte une interface permettant d'abstraire les accès directs à Git, afin que l'ensemble des greffons développés dans l'outil CanHOE2 puissent utiliser les commandes natives de Git sans se soucier de leur implémentation. Nous nous intéressons dans cette partie à l'organisation des modèles du système en développement au sein de dépôts Git.

```
Racine/
|--- CanHOE2_Projects/
    |--- Projet1/
    |--- Projet2/
        |--- Requirements Analysis/
        |    |--- system.hoe2-ra
        |    |--- UC-2014_07_17_10am_06m_30s_000124micros/
        |    |    |--- 2014_07_17_10am_13m_01s_150543micros.hoe2-ra-sc
        |    |    |--- 2014_07_17_10am_13m_45s_421354micros.hoe2-ra-sc
        |    |    |--- 2014_07_17_10am_12m_31s_100523micros.hoe2-ra-sc
        |    |    |--- 2014_07_17_10am_14m_23s_216435micros.hoe2-ra-sc
        |    |--- UC-2014_07_17_10am_14m_30s_002134micros/
        |         |--- 2014_07_17_10am_15m_43s_012345micros.hoe2-ra-sc
        |--- System Analysis/
        |--- System Design/
        |--- System Implementation/
        |--- system.hoe2-project
        |--- Tasks/
            |--- 2014_07_16_06am_06m_30s_000123micros.hoe2-tasks
```

Listing 9.1 – Organisation des développements sous Git

Un dépôt Git pour chaque projet / système considéré. Nous avons fait le choix d'un projet et d'un dépôt indépendant pour chaque système considéré (application ou plate-forme). Pour chaque dépôt, nous avons scindé et organisé les éléments, tel qu'illustré dans le listing 9.1. Dans chaque dépôt, nous retrouvons quatre dossiers pour chacune des phases du processus $\langle \mathsf{HOE} \rangle^2$. Dans la phase d'analyse du besoin, le modèle est lui-même scindé en plusieurs fichiers.

14. Un commit fige le dépôt dans un état de développement stable. Les possibles traductions de *sauvegarde*, ou *capture* sont assez confuses. Aussi, nous avons choisi, pour le reste de ce manuscrit de conserver le terme anglais *commit*.

Le fichier *system.hoe2-ra* contient la modélisation du système, des cas d'utilisation, des acteurs et des différentes associations entre les acteurs et le cas d'utilisation. Pour chaque cas d'utilisation, l'ensemble de ses scénarios est stocké dans un dossier portant l'identifiant du cas d'utilisation. L'identifiant du cas d'utilisation est constitué à partir de la date et de l'heure à laquelle il a été créé. Chaque scénario est enregistré dans un fichier unique, dont le nom est également obtenu à partir de la date à laquelle il a été produit. Ce choix de découpage a été réalisé afin de pouvoir aisément référencer les scénarios pour les phases suivantes.

Pour chaque dépôt, un fichier *system.hoe2-project*, contient l'ensemble des informations du projet. Il s'agit d'un modèle conforme au méta-modèle de gestion de projet présenté dans le chapitre précédent. Un cinquième dossier contient la liste des feuilles de tâche du projet. Chaque feuille de tâche est sauvegardée dans un fichier indépendant, et référence des objets du modèle. Les feuilles de tâche sont assignées à un développeur dans le modèle *system.hoe2-project*.

Développement courant maintenu par le chef de projet. Le développement courant maintenu par le chef de projet dans la *branche principale* du dépôt Git qualifie l'état dans lequel repose le système modélisé. La branche principale intègre toutes les tâches de modélisation des développeurs qui ont été validées par le chef de projet. Cette branche n'est accessible et maintenue que par le chef de projet. Dans cette branche sont contenues les tâches affectées aux différents développeurs et la modélisation courante du système.

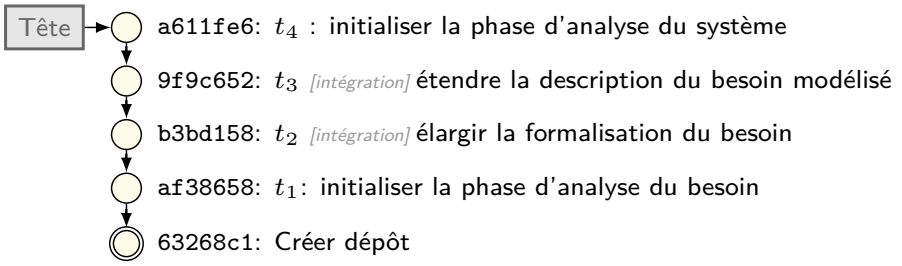


FIGURE 136 – Branche courante du chef de projet

La figure 136 illustre l'arbre des commits dans la branche courante du chef de projet. Chaque cercle correspond à un commit. L'identifiant et le label du commit le définissent. Les commits sont organisés dans le temps. La lecture s'effectue de bas en haut, le premier commit est le commit initial, souvent associé à la création du dépôt. Le dernier commit se nomme *Tête*. Il distingue le tout dernier état dans lequel se trouve un développement dans un état stable.

Dans le cas de la branche principale du chef de projet, chaque commit correspond soit à l'intégration de la tâche de modélisation d'un développeur, soit à une tâche automatique d'initialisation ou de clôture de phases réalisée par le chef de projet.

Organisation des branches. Nous avons fait le choix d'organiser les différentes branches des dépôts Git en respectant le schéma suivant. Chaque branche désigne un développeur particulier. Chacune de ces branches est elle-même divisée en sous-branches, correspondant aux différentes tâches assignées aux développeurs. Dans la sous-branche est contenue la feuille de tâche ainsi que l'ensemble des fragments de modèles nécessaires pour réaliser la tâche de modélisation. Dans le cas où d'autres feuilles de tâches sont référencées, elles sont également importées. Néanmoins,

si ces autres feuilles de tâches sont assignées à d'autres développeurs, seule la feuille de tâche et non le modèle associé seront accessibles par le développeur.

Lorsqu'une tâche est initialisée et assignée à un développeur, une sous-branche est donc créée et le développeur en est alerté. Il peut alors changer de branche afin de s'occuper du développement demandé. Une fois le développement terminé et validé, la sous-branche du développeur est fusionnée avec la branche du chef de projet.

```
$ git checkout -b <ID_DEVELOPER> ‘git rev-list --max-parents=0 HEAD‘
```

Listing 9.2 – Création d'une branche pour le développeur à partir du commit initial

Les commandes présentées dans les listings 9.2 et 9.3 illustrent les commandes Git nécessaires à la création d'une nouvelle branche pour le développeur et de sous-branches. Dans la première, le choix a été fait de créer la sous-branche à partir du point d'origine du dépôt afin de ne pas tirer tout l'historique de développement effectué jusque-là.

```
$ git checkout <ID_DEVELOPER>
$ git checkout -b <ID_TASK>
```

Listing 9.3 – Création d'une sous-branche pour chaque tâche

La figure 137 résume l'organisation des branches et des sous-branches. Lorsque le chef de projet décide d'ajouter un développeur à un projet, une nouvelle branche est créée. Pour chaque tâche assignée, une sous-branche dans la branche du développeur est créée (listing 9.3).

Sous-modules et dépendances entre les dépôts. Lors de la phase de conception du système, les plates-formes du système en cours de développement sont identifiées. Les modèles

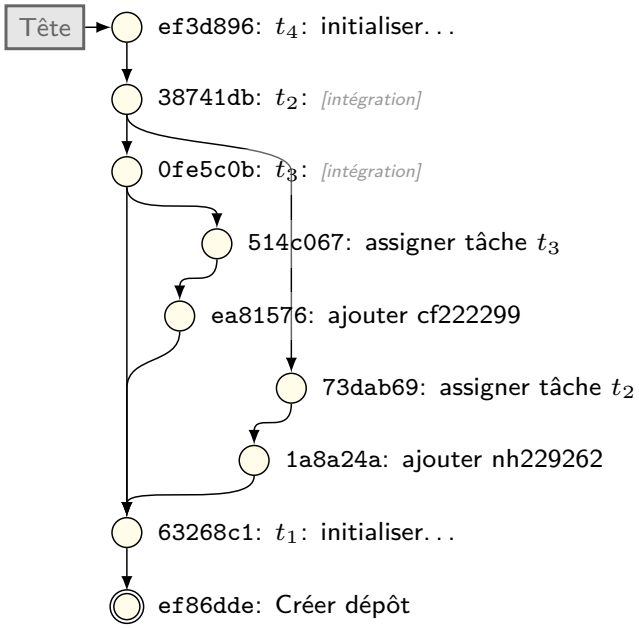


FIGURE 137 – Intégration des tâches de modélisation du point de vue des branches

de ces plates-formes sont développés dans des dépôts séparés. Les dépendances entre les dépôts sont assurées par les *sous-modules Git*. Les sous-modules ont permis de scinder le développement de systèmes entre différents dépôts. Nous avons pour cela fait le choix de récupérer la branche principale des sous-modules afin de récupérer la version courante des plates-formes identifiées.

Les sous-modules Git sont parfaitement adaptés pour ce cas d'utilisation. Ils permettent de séparer les développements dans des dépôts distincts et indépendants, avec un historique propre à chaque dépôt. Les sous-modules peuvent être récursifs afin de prendre en compte plusieurs niveaux de composition de plates-formes.

Intégration transparente de Git. S'il est possible d'utiliser les commandes définies au-dessus pour organiser et gérer les versions des développements, l'une des forces de l'outil CanHOE2 réside dans l'intégration discrète de Git au sein de l'outil. Le greffon *canhoe2.git* est le point d'entrée nécessaire pour le dialogue avec Git. Il définit tous les échanges entre l'outil et Git afin de pouvoir réaliser toutes les activités d'assignation de tâches, de création de branches, de changement entre les différentes branches, etc. Ainsi, le travail de modélisation n'est pas perturbé par un travail supplémentaire de gestion de version à la main.

En résumé

- ✓ *Intégration transparente de Git*
- ✓ *Organisation efficace du code de chaque système dans des dépôts séparés*
- ✓ *Organisation des développeurs et des tâches en branches et sous-branches*

Synthèse du chapitre

Dans ce chapitre, nous avons présenté l'outillage CanHOE2. Cet outillage permet un développement multi-utilisateurs et supporte aujourd'hui partiellement la méthode $\langle \mathsf{HOE} \rangle^2$ selon trois axes : processus, langage et gestion de projet. Nous avons également cherché à améliorer l'utilisation de l'outils en ciblant des enjeux industriels tels que la génération de documentation et de code, ainsi que l'organisation du code dans des dépôts de versions, sans pour autant alourdir le travail du développeur par une mauvaise connaissance ou une mauvaise manipulation des outils de gestion de versions.

Du point de vue technique, nous avons privilégié le langage Java et l'environnement de développement Eclipse. Eclipse fournit un environnement de développement modulaire et une infrastructure personnalisable basée sur la conception de greffons. L'outillage CanHOE2 a été structuré en différents greffons venant personnaliser cet environnement pour l'adapter à la méthode $\langle \mathsf{HOE} \rangle^2$. Nous avons utilisé l'outil Papyrus MDT et le projet UML2 pour la définition des méta-modèles de la méthode et Graphiti pour la conception des éditeurs graphiques des modèles $\langle \mathsf{HOE} \rangle^2$. Pour l'intégration totalement transparente de Git au sein de l'outil CanHOE2, nous avons développé un greffon qui exporte des interfaces Java très primitives pour utiliser les commandes Git.

Chapitre 10

Validation

Publication pertinente à ce chapitre :

[**Hili *et al.* 2014b**] Nicolas Hili, Christian Fabre, Sophie Dupuy-Chessa, Dominique Rieu et Ivan Llopard. *Model-Based Platform Composition for Embedded System Design.* In IEEE 8th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC-14) (IEEE MCSoC-14), Aizu-Wakamatsu, Japan, Septembre 2014

Liste des sections

Ce chapitre permet de valider certaines propriétés de la méthode $\langle \mathsf{HOE} \rangle^2$ à partir du cas d'étude du *système de suivi de passants*. Il a pour vocation de présenter toutes les étapes de développement du système (ces dernières ont été présentées dans les chapitre 6 et 7), mais détaille les deux dernières activités du processus : la *transformation successive des modèles de l'application* pour illustrer la composition et l'injection successive dans les conteneurs (cf. fig. 79 de la page 134) et la *génération de code* pour valider l'implémentation du système sur des plates-formes réelles.

Nous allons détailler l'étude de cas de la manière suivante : la section 1 décrit le cas d'étude, la réalisation technique ainsi que les propriétés que nous souhaitons valider ; les sections 2 et 3 présentent les résultats obtenus en termes de transformation de modèles et de génération de code. Ce chapitre fournit une version étendue des résultats initialement présentés dans [Hili *et al.* 2014b].

1 Étude de cas et formulation des propriétés à valider

Cette première section présente le cas d'étude réalisé. Elle est structurée en trois parties. La première partie détaille le cas d'étude ; la seconde présente le matériel utilisé et le montage réalisé ; la dernière décrit les propriétés que nous allons vérifier.

1.1 Présentation du cas d'étude

L'étude de cas met en scène deux plates-formes assemblées permettant le pilotage d'une caméra fixée sur un support orientable. Cette caméra transmet un flux vidéo qui est analysé pour connaitre la position d'un visage d'un passant. D'autres périphériques sont disponibles afin de permettre l'implémentation de l'application du *système de suivi de passants* illustrée dans les précédents chapitres.

Les points de départ de ce chapitre sont les *modèles de conception et d'implémentation du système de suivi de passants*. Ces modèles diffèrent de ceux initialement réalisés pour illustrer les activités de distribution et d'implémentation dans le chapitre 6 (cf. fig. 70b de la page 120 et fig. 74a de la page 126). Cette modélisation ne tenait pas compte de la composition de plates-formes détaillée dans le chapitre 7. Aussi, nous proposons dans cette partie de nous appuyer sur les modèles présentés dans les figures 138 et 139.

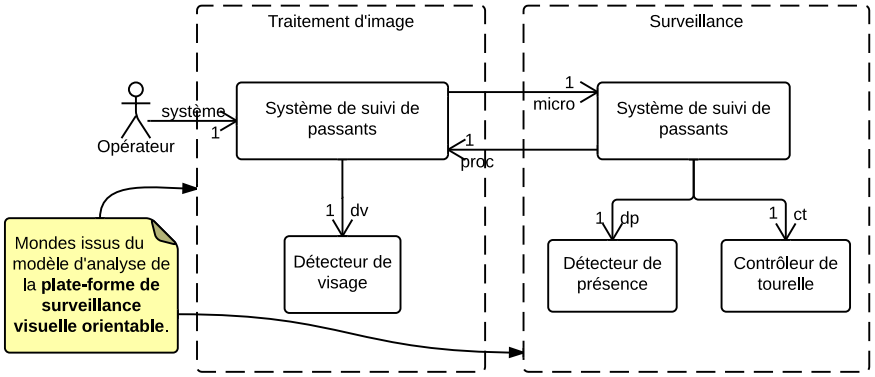


FIGURE 138 – Modèle de conception du système de suivi de passants

Le modèle de la figure 138 est le modèle de *conception du système de suivi de passants* sur la *plate-forme de surveillance visuelle orientable*. Le modèle d'*analyse de la plate-forme de surveillance visuelle orientable* a été présenté dans le chapitre 7 (cf. fig. 90 de la page 146). Ce dernier modèle contient les mondes, qui ont été utilisés pour héberger les différents objets du modèle d'implémentation de la figure 138.

Nous avons ensuite développé le modèle d'*implémentation du système de suivi de passants* en *injectant* les différents objets du système dans les conteneurs de la plate-forme. L'activité d'injection dans les conteneurs est représentée par la figure 139. Ces deux modèles constituent le point d'entrée pour valider différentes propriétés sur la méthode. Nous n'aborderons donc les modèles qu'à partir de la phase d'implémentation du système et nous partirons du modèle d'implémentation de la figure 139 afin d'illustrer, dans un premier temps, l'impact de la ré-écriture successive des machines à états sur les objets applicatifs du système de suivi de passants

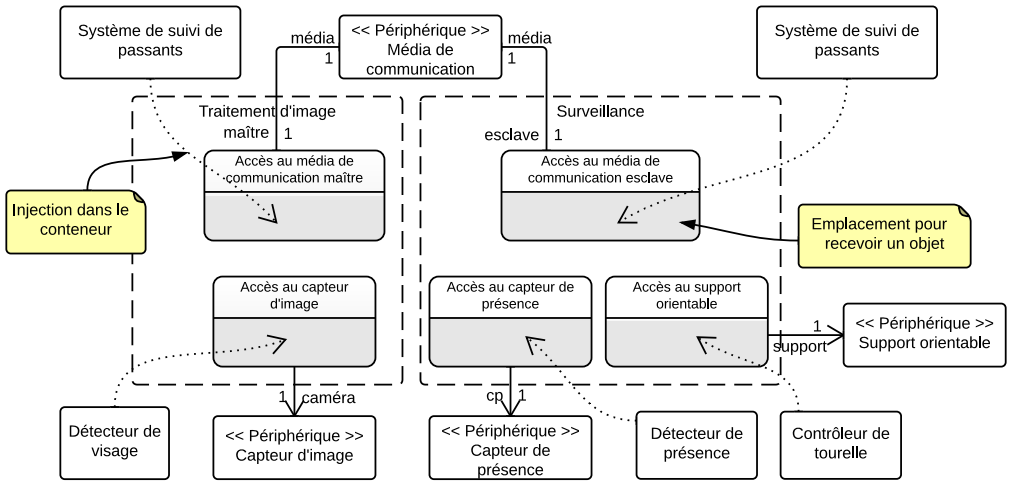
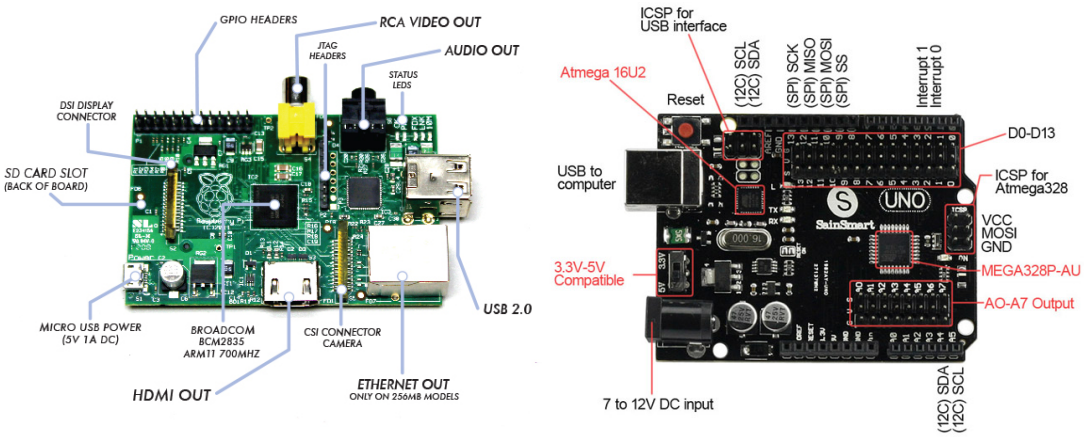


FIGURE 139 – Modèle d'implémentation du *système de suivi de passants*

et dans un second temps, la génération de code à partir du modèle implémenté sur les *plates-formes Arduino UNO et Raspberry PI.*

1.2 Réalisation technique



(a) Raspberry PI

(b) Arduino UNO

FIGURE 140 – Photos des deux plates-formes à assembler

Matériel utilisé. La génération de code qui sera présentée en section 3 s'appuie sur les deux plates-formes finales Raspberry PI et Arduino UNO. Ces deux plates-formes sont illustrées dans la figure 140. La plate-forme Raspberry PI héberge le système d'exploitation Linux et dispose à ce titre d'un ensemble de commandes déjà programmées. Elle possède un certain nombre d'interfaces pour le pilotage de différents capteurs et actionneurs. Elle possède notamment deux ports USB, un port HDMI pour le branchement d'un écran, un connecteur SD CARD

pour le système et les données, un connecteur de caméra au format Camera Serial Interface (CSI), un port Ethernet. La plate-forme possède également vingt-six broches numériques. Ces broches permettent entre autres de se connecter à une autre plate-forme en utilisant les bus I^2C et SPI. Il est également possible de connecter un faible nombre de périphériques dessus. Le développement sur cette plate-forme est traditionnellement réalisé en Python ou en C.

La plate-forme Arduino UNO permet de contrôler plusieurs périphériques connectés aux broches analogiques et numériques. Chaque broche peut être configurée en entrée (capteurs) ou en sortie (actionneurs). La plate-forme possède également différentes interfaces (I^2C, SPI et USB). Elle est alimentée par USB depuis un ordinateur ou bien par une alimentation externe. Le développement sur cette plate-forme s'effectue au moyen du langage dédié Arduino.

Nous avons utilisé pour les besoins de cette étude de cas différents périphériques tels qu'un capteur de présence PIR pour la détection de présence, un support orientable par deux servomoteurs, une caméra embarquée pour la détection de visage. Les deux plates-formes sont connectées en USB et I^2C. Nous avons fait le choix de rendre le bus USB inaccessible pour l'application modélisée et de ne l'utiliser que comme ressource interne des deux plate-formes. Du côté de la plate-forme Raspberry PI, le bus USB permet l'installation d'une application sur la plate-forme Arduino UNO et son redémarrage. Le bus I^2C est quant à lui accessible depuis l'application hébergée sur les deux plates-formes. Elle permet aux deux parties de l'application de pouvoir communiquer et transmettre des données en série.

Réalisation du montage. La figure 141 illustre le montage des deux plates-formes et du support pour la caméra. Nous avons connecté les deux cartes par l'intermédiaire d'une platine d'expérimentation. Les deux plates-formes sont connectées avec les protocoles I^2C et USB. La caméra embarquée et connectée via la broche CSI du Raspberry PI et est montée sur la tourelle. Nous avons également installé une LED pour prévenir en cas de détection (cet exemple sera utilisé pour la génération du code). La plate-forme Raspberry PI est directement alimentée depuis un ordinateur et connectée à ce dernier via son port Ethernet. La plate-forme Arduino

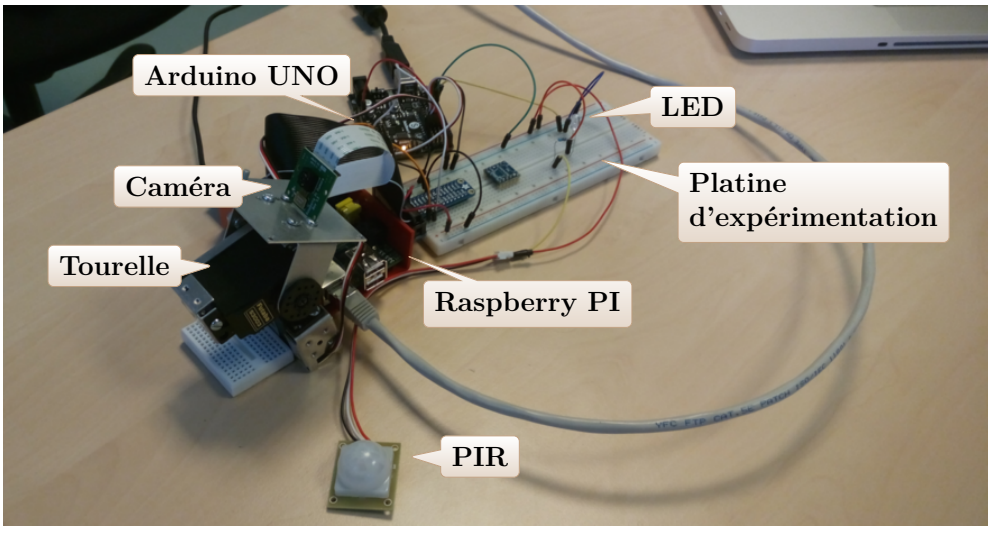


FIGURE 141 – Réalisation du montage

UNO est quant à elle alimentée par une alimentation externe. Enfin, la plate-forme Raspberry PI est connectée à un écran permettant de simuler les réponses du système à l'*opérateur*.

1.3 Formulation des propriétés à valider

Nous avons formulé plusieurs propriétés qui doivent être validées par cette étude de cas.

Propriété 1 (Génération de code compilable et sans erreur) Le cas d'étude doit montrer qu'il est possible, après avoir défini un générateur spécifique pour une plate-forme, de générer du code pour cette plate-forme cible. La génération du code doit pouvoir se faire à partir d'un modèle d'implémentation d'un système sur les différents mondes de la plate-forme. Le code produit ne doit pas générer d'erreur à la compilation et à l'exécution. En outre, son exécution doit correspondre au comportement modélisé du système.

Propriété 2 (Code optimisé pour la plate-forme) Les plates-formes de systèmes embarqués possédant des ressources limitées, en particulier en terme de capacité des mémoires permanentes et volatiles, le cas d'étude doit montrer qu'il est possible de générer un code optimisé pour la plate-forme en question.

Propriété 3 (Composition de plates-formes) Le cas d'étude doit montrer qu'il est possible d'appliquer les deux types de composition de plates-formes avec plusieurs niveaux de plates-formes sur un exemple. L'injection successive des objets applicatifs dans les différents conteneurs doit permettre de produire un modèle *valable* d'implémentation du système sur les plates-formes finales. Par *valable*, nous entendons que le modèle produit ne nécessite aucune modification manuelle et puisse directement servir à générer du code conforme au comportement initialement modélisé du système.

Propriété 4 (Accès aux ressources et périphériques de la plate-forme) Le cas d'étude doit montrer que l'activité d'injection dans les conteneurs de la plate-forme et l'exécution des règles d'implémentation des conteneurs produisent bien un modèle d'implémentation du système dans lequel les objets applicatifs accèdent désormais aux ressources et périphériques modélisés provenant du modèle d'analyse de la plate-forme. Cette propriété doit être validée au niveau de la modélisation d'une part et à l'exécution d'autre part, en générant le code du système à partir du modèle d'implémentation obtenu et en l'exécutant sur la plate-forme réelle.

Propriété 5 (Langage commun pour l'application et la plate-forme) Le cas d'étude doit montrer que les modèles utilisés pour la modélisation d'objets applicatifs et de la plate-forme sont conformes au même langage. En particulier, le code généré par l'approche doit être conforme aux cas d'utilisation modélisés pour l'application et la plate-forme.

En résumé

- ✓ *Introduction du cas d'étude*
- ✓ *Présentation du matériel utilisé et du montage technique*
- ✓ *Formulation de cinq propriétés que nous souhaitons valider au travers de ce cas d'étude*

2 Composition de plates-formes

Cette section illustre l'application de la composition de plates-formes et l'impact sur la réécriture du comportement des objets applicatifs. Nous montrons dans cette section comment l'injection successive dans les conteneurs des plates-formes permet de produire un modèle d'implémentation du système de suivi de passants sur les plates-formes finales, dans lequel les objets du système accèdent désormais aux ressources et périphériques de la plate-forme. Pour montrer l'application de la composition et ainsi valider les propriétés 3, 4 et 5 sur le cas d'étude, nous commençons dans cette section par présenter la définition d'une règle d'implémentation.

Une fois cette définition introduite, nous détaillons, en partant du modèle d'*implémentation du système de suivi de passants* sur la *plate-forme de surveillance visuelle orientable* 139 l'impact de ces règles à tous les niveaux pour produire un modèle d'implémentation du système de suivi de passants sur les plates-formes finales Arduino UNO et Raspberry PI.

2.1 Écriture des règles d'implémentation

Nous avons défini pour chaque conteneur des plates-formes les règles d'implémentation permettant l'injection d'un objet. Ces règles ont été écrites avec l'outil Eclipse Wizard Language (EWL), faisant partie de la plate-forme Eclipse MDT. Elles ont vocation à être intégrées au sein de l'outil CanHOE2.

EWL permet de transformer un modèle « en place ». Le listing 10.1 illustre la définition d'un *assistant* EWL. Un assistant permet de définir des règles qui peuvent s'appliquer sur un modèle pour le modifier. Il permet l'ajout, la modification où la suppression d'éléments. Il s'intègre parfaitement à des éditeurs arborescents EMF ainsi qu'à des éditeurs graphiques. L'approche est différente de la production d'un modèle à partir d'un autre tel que ATL ou ETL. Elle est proche du mode de raffinement de ATL. La syntaxe du langage EWL est basée sur le langage OCL et Java.

La règle d'implémentation associée à chaque conteneur d'une plate-forme est composée de trois étapes, deux étant génériques à tous les conteneurs et une troisième spécifique.

étape 1 (générique) : Associer à l'objet injecté dans le conteneur tous les périphériques et ressources auxquels le conteneur donne accès ;

étape 2 (générique) : Assurer l'associativité de l'injection, dans le cas où la plate-forme est obtenue par composition (i.e. les conteneurs de la plate-forme sont injectés dans les conteneurs de la ou les plates-formes incrémentées / assemblées) ;

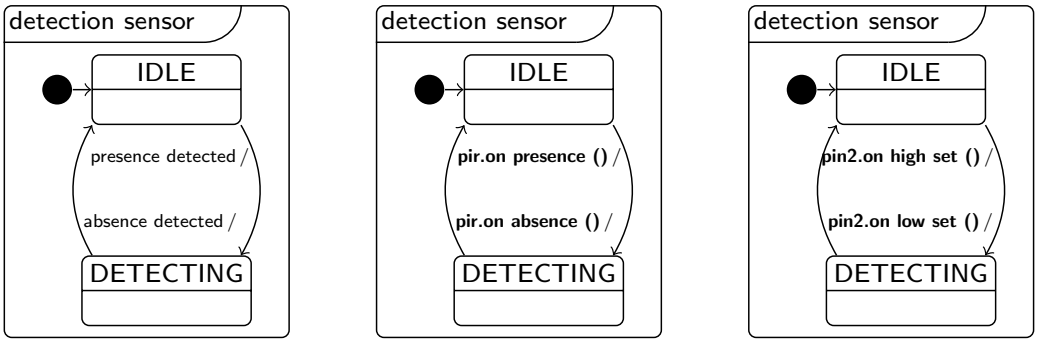
étape 3 (spécifique) : Ré-écrire le comportement de la machine à états pour communiquer directement avec les ressources et périphériques de la plate-forme.

Règle 1 (Accès au capteur de présence) Pour tous les *signaux* de la machine à états de l'objet :
→ Le signal *« presence detected »* devient *« pir.on presence () »*
→ Le signal *« absence detected »* devient *« pir.on absence () »*
→ Tous les autres signaux restent inchangés

```
wizard Object2ImplementedObject {
  guard : self.isKindOf(Object) and self.packagedElement->select(pe | pe.
    oclIsKindOf(Injection))->notEmpty()

  title : "Implement " + self.name

  do {
    [...]

    for (r : Region in objectBehavior.regions) {
      for (t : Transition in r.transitions) {
        if (t.trigger <> null) {
          if (t.trigger.signal.name = 'presence detected') {
            t.trigger.signal = presenceSignal;
            t.trigger.kind = TriggerKind#on;
            t.trigger.sender = pirObjectPeripheralAssociation.role;
          }
          else if (t.trigger.signal.name = 'absence detected') {
            t.trigger.signal = absenceSignal;
            t.trigger.kind = TriggerKind#on;
            t.trigger.sender = pirObjectPeripheralAssociation.role;
          }
        }
      }
    }
  }
}
```

Listing 10.1 – Exemple de règle définie en EWL

La règle 1 présentée listing 10.1 illustre la règle d'implémentation du conteneur d'accès au périphérique de présence de la *plate-forme de contrôle de tourelle PT* qui permet de ré-écrire l'objet Détecteur de présence. La figure 142 illustre une implémentation successive de cet objet sur la plate-forme Arduino UNO. Dans cette implémentation, l'objet initial (cf. fig. 142a) est d'abord implémenté sur la *plate-forme de contrôle de tourelle PT* (cf. fig. 142b) par l'exécution des règles d'implémentation. Il est ensuite implémenté sur la plate-forme *Arduino UNO* (cf.



(a) Avant la première ré-écriture (b) Première ré-écriture (c) Seconde ré-écriture

FIGURE 142 – Détecteur de présence implémenté dans le conteneur d'accès au PIR lui-même implémenté dans le conteneur d'accès à une broche

fig. 142c). L'objet était initialement indépendant de la plate-forme et son injection dans le conteneur a permis de le modifier pour l'implémenter sur la plate-forme.

Les règles d'implémentation des conteneurs venant d'être présentées, la partie suivante présente le résultat de l'implémentation de l'application du *système de suivi de passants* sur les différentes plates-formes composées.

2.2 Application au système de suivi de passants

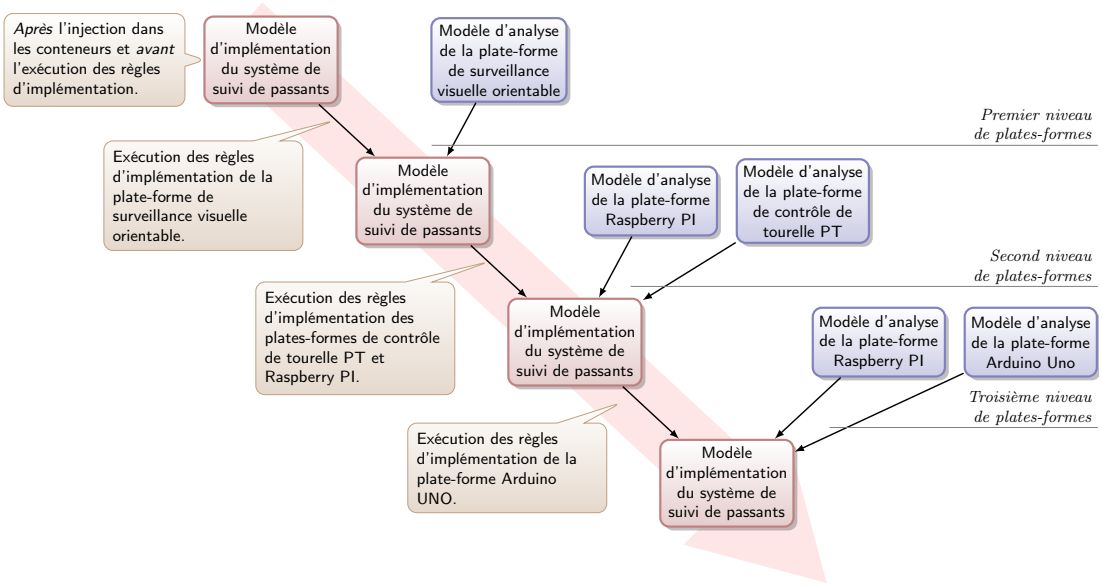


FIGURE 143 – Résumé des modèles d'implémentation produits

Nous présentons dans cette partie l'injection successive dans les conteneurs des différents niveaux de plates-formes pour produire le modèle d'*implémentation du système de suivi de passants* sur les deux plates-formes finales Arduino UNO et Raspberry PI. La figure 143 résume les différents modèles d'implémentation que nous allons obtenir dans cette partie. Le modèle d'implémentation du système initial est illustré dans la figure 139 de la page 211. Nous détaillons dans cette partie chaque modèle d'implémentation obtenu suite à l'exécution des règles d'implémentation, ainsi que l'impact des règles sur la ré-écriture du comportement des objets. Cet impact sera illustré sur le comportement de l'objet Contrôleur de support du système.

Implémentation sur la plate-forme de surveillance visuelle orientable. En prenant en compte la *plate-forme de surveillance visuelle orientable* comme plate-forme de plus haut niveau dans la composition, la première injection dans les conteneurs de la plate-forme produit le modèle implémenté du système de *suivi de passants* où tous les objets sont ré-écrits et accèdent désormais aux périphériques et ressources de la plate-forme. La figure 144 illustre cette première implémentation. Les objets ayant été affectés par les règles d'implémentation des conteneurs ont été illustrés en gras. Par exemple, les deux fragments du système ne communiquent désormais

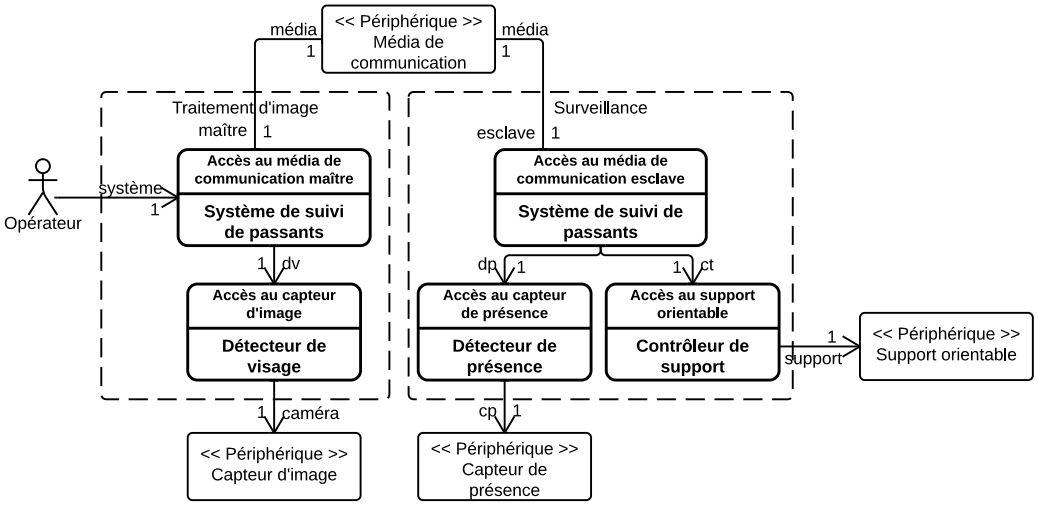
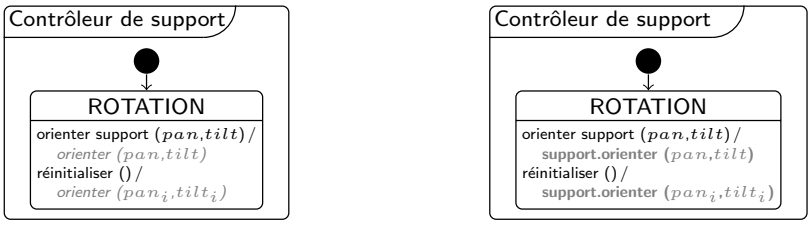


FIGURE 144 – Modèle d'implémentation du système sur la *plate-forme de surveillance visuelle orientable*

plus ensemble directement, mais par l'intermédiaire d'un Média de communication, tandis que le Contrôleur de support accède désormais à un Support orientable.



(a) Avant implémentation (b) Première ré-écriture

FIGURE 145 – Première ré-écriture du comportement du contrôleur de support

La figure 145 illustre le comportement du Contrôleur de support avant implémentation (fig. 145a) et après ré-écriture (fig. 145b). Initialement, son comportement permettait de réceptionner les messages *orienter support (pan, tilt)* et *réinitialiser ()* provenant du système. Il pouvait alors envoyer le signal *orienter ()* en fournissant les deux angles reçus lors de la réception du message ou deux angles initiaux dans le cas du message *réinitialiser ()*. Après implémentation de l'objet dans le conteneur d'accès au support, un nouveau message permet d'initialiser le système dans la position initiale. Ensuite, il est capable de dialoguer directement avec le périphérique Support orientable au moyen des messages *support.orienter (pan,tilt)* et *support.orienter (pan_i,$tilt_i$)*.

Implémentation sur les plates-formes de contrôle de tourelle PT et Raspberry PI. La figure 146 illustre l'implémentation du *système de suivi de passants* sur les *plates-formes de contrôle de tourelle PT* et *Raspberry PI*. Dans ce modèle, les deux fragments du système

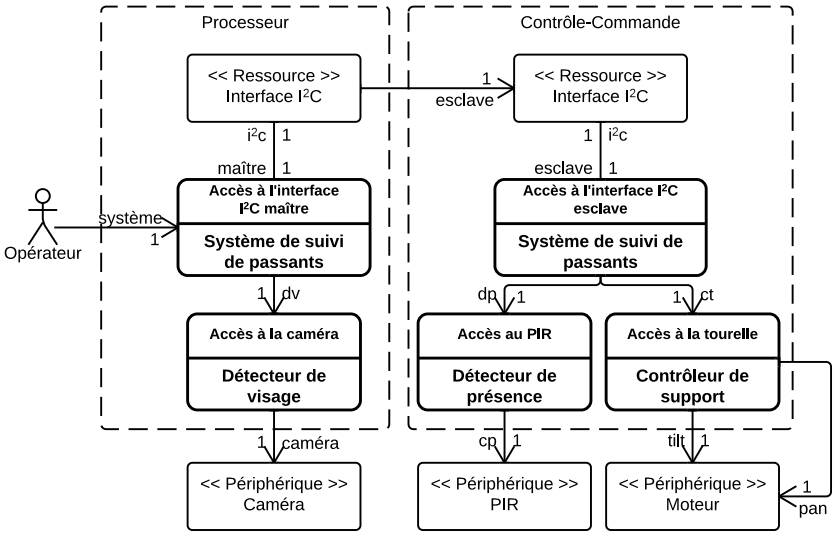


FIGURE 146 – Modèle d'implémentation du système sur les plates-formes de contrôle de tourelle PT et Raspberry PI

ne dialoguent désormais plus au travers d'un média de communication mais d'interfaces I^2C. Le premier fragment est pour cela implémenté dans un conteneur d'accès à l'interface en mode *maître* tandis que le second fragment est implémenté en mode *esclave*. Le contrôleur de support n'a désormais plus accès à un support mais directement aux deux moteurs *pan* et *tilt* permettant d'orienter la caméra dans deux directions. Quant aux détecteurs de présence et de visage, ils accèdent désormais aux périphériques caméra et PIR.

La figure 147 illustre la seconde ré-écriture du Contrôleur de support accédant désormais aux deux moteurs. Deux régions ont été créées afin de permettre le contrôle séparé des deux moteurs. La région principale permet toujours de recevoir les messages *orienter support ()* et *réinitialiser ()*. Cependant, l'effet de la transition permet de modifier les valeurs des pulsations des deux moteurs afin de les contraindre en position.

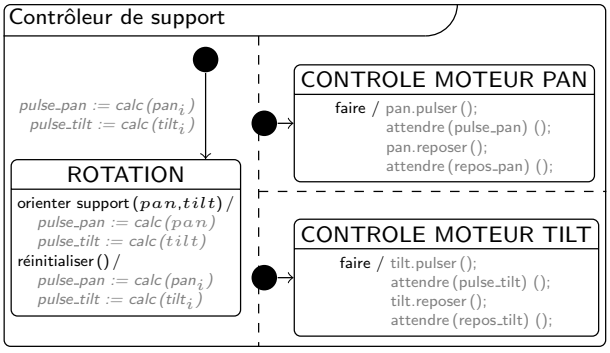


FIGURE 147 – Seconde ré-écriture du comportement du contrôleur de support

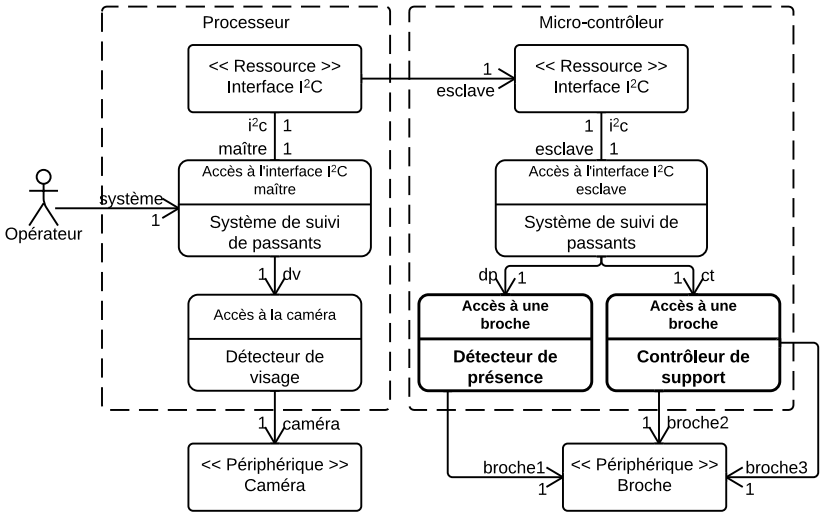


FIGURE 148 – Modèle d'implémentation du système sur les plates-formes Arduino UNO et Raspberry PI

Implémentation sur les plates-formes Arduino UNO et Raspberry PI. La figure 148 illustre l'implémentation du *système de suivi de passants* sur les *Arduino UNO* et *Raspberry PI*. Les objets hébergés sur la *plate-forme Raspberry PI* ont déjà atteint leurs implémentations finales, ils ne sont donc pas impactés par ce dernier niveau. Les objets hébergés dans le monde Micro-contrôleur de la *plate-forme Arduino UNO* sont quant à eux ré-écrits. Ils sont désormais connectés aux broches de la plate-forme. Dans la figure 149, les deux moteurs sont désormais remplacés par deux broches de la plate-forme Arduino UNO. Le schéma général de la machine à états ne varie pas, seuls les messages *pulser ()* et *reposer ()* sont remplacés par le changement de l'état des deux broches.

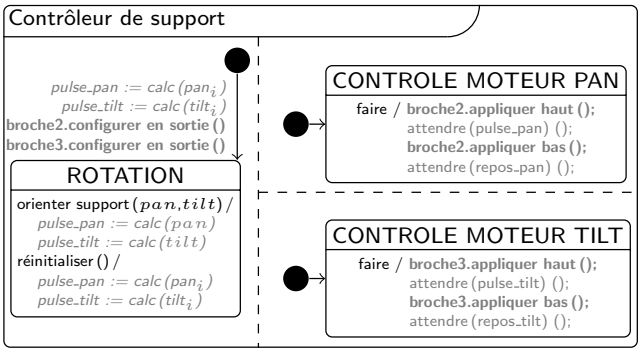


FIGURE 149 – Troisième ré-écriture du comportement du contrôleur de support

Résultats obtenus et propriétés validées. L'injection successive dans les conteneurs des plates-formes à tous les niveaux a permis de montrer qu'à partir d'une première décision d'in-

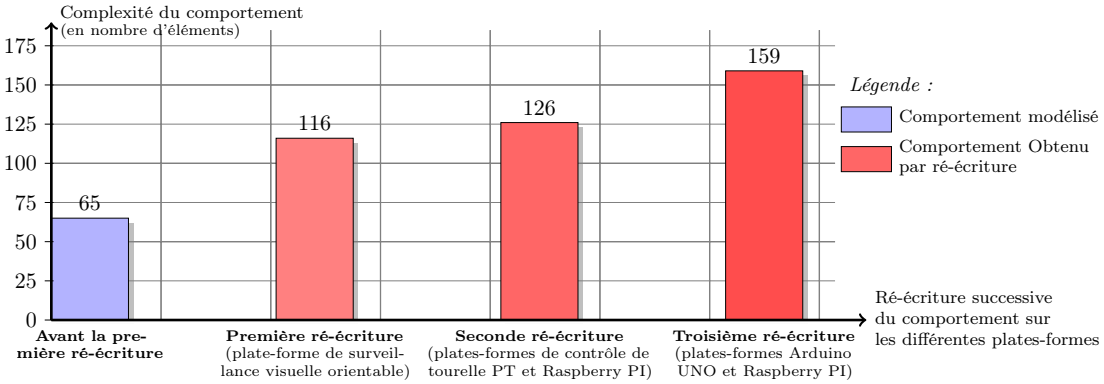


FIGURE 150 – Évolution de la complexité des machines à états suivant l'injection successive dans les plates-formes

génierie qui consistait à sélectionner les conteneurs dans lesquels seraient injectés les objets de conception du système, il était possible de dérouler la composition de plates-formes pour obtenir une implémentation du système sur les plates-formes finales. Ce résultat a pu être obtenu en utilisant les différentes règles d'injection dont la structure a été présentée dans la partie précédente. Le dernier modèle d'implémentation peut être utilisé pour générer le code du système sur les plates-formes, conforme au comportement initial.

La figure 150 illustre l'évolution de la complexité des machines à états suivant les différents niveaux de plates-formes. Nous entendons par complexité le nombre de concepts nécessaires (région, états, transitions, événements déclencheurs, etc.) à la modélisation des comportements des objets implémentés. L'axe des abscisses présente les différentes implémentations. L'axe des ordonnées présente la complexité. Nous pouvons remarquer l'évolution progressive de la complexité en fonction des différents niveaux d'implémentation. Cette figure laisse entrevoir les bénéfices de l'injection sur plusieurs niveaux intermédiaires de plates-formes afin de permettre à un développeur de modéliser une application très abstraite entièrement découplée de la plate-forme réelle d'implémentation, la complexité due à la plate-forme dans le modèle final n'étant introduit que graduellement et de manière structurée.

Le tableau. 41 résume les propriétés validées. Nous avons défini des règles d'implémentation au format EWL et avons montré leurs impacts sur la transformation des comportements des objets. L'implémentation permet de transformer un objet indépendant de la plate-forme en un objet propre à celle-ci, lui permettant d'utiliser les ressources et périphériques de cette

Hypothèses validées	
Hypothèse 1 (Génération de code compilable et sans erreur)	
Hypothèse 2 (Code optimisé pour la plate-forme)	
Hypothèse 3 (Composition de plates-formes)	✓
Hypothèse 4 (Accès aux ressources et périphériques de la plate-forme)	✓
Hypothèse 5 (Langage commun pour l'application et la plate-forme)	✓

TABLE 41 – Hypothèses validées par la composition de plates-formes

dernière. Cela valide les hypothèses 4 et 5. L'injection successive jusqu'à une implémentation finale permet de valider l'hypothèse 3. Nous avons également montré que les règles d'injection dans les conteneurs s'appliquaient à la fois sur les modèles de l'application et de la plate-forme, validant ainsi l'hypothèse 5.

Les figures 143 et 150 laissent entrevoir l'intérêt de la composition de plates-formes et de l'injection successive dans les conteneurs. Pour la génération de code sur les *plates-formes Arduino UNO et Raspberry PI*, seul le dernier modèle d'implémentation nous intéresse. Ce dernier modèle est toutefois trop compliqué pour être modélisé car très spécifique à la plate-forme et il est préférable de n'avoir à modéliser que le premier modèle de la chaîne et de laisser à un processus automatique le soin de produire les différents modèles pour atteindre le modèle le plus concret en vue de générer le code. L'injection successive dans les conteneurs, due à l'associativité de l'injection offre cette possibilité.

En résumé

- ✓ *Composition de plates-formes illustrée sur le système de suivi de passants*
- ✓ *Définition de deux règles génériques pour l'injection successive dans les conteneurs*
- ✓ *Définition d'une règle spécifique au conteneur permettant de ré-écrire le comportement d'un objet*

3 Génération du code des plates-formes

Cette section présente nos résultats en terme de génération de code. Ces résultats ont pour but de valider les propriétés 1, 2, 4 et 5. Afin d'illustrer simplement la génération de code, nous nous appuyons dans cette section sur un modèle plus simple que ceux présentés dans les chapitres précédents. Ce modèle, illustré par la figure 151 est celui d'un *système de détection de présence* basé sur la *plate-forme Arduino UNO*. Nous avons fait le choix de présenter la génération de code uniquement sur la plate-forme Arduino UNO pour la faible quantité de ressources embarquées dans cette plate-forme, par rapport à la plate-forme Raspberry PI. Le processeur de la plate-forme Arduino UNO est un *ATmega328*. Il possède une mémoire vive *SRAM*[1] de 2048 octets[2]. Il possède également une mémoire flash de 16ko pour stocker le code de l'application. Cette faible quantité nous permet de justifier le besoin d'optimiser le code généré (propriété 2) pour être adapté à la plate-forme. Puisque le langage ⟨HOE⟩[2] est basé sur le paradigme d'envoi de messages[3] [Obermaisser 2004], nous présentons dans cette section la génération de code pour l'ensemble des composants de ce paradigme. La première partie introduit le système modélisé. Les parties suivantes abordent la génération de code des différents éléments.

Le langage utilisé pour la génération est le langage Acceleo. Il permet de définir des schémas de génération de code. Certains listings présentant la génération de code sont présentés dans cette parties. Les autres listings sont fournis en annexe.

1. En anglais, Static Random Access Memory (SRAM).
2. Une autre version est équipée d'un processeur *ATmega168* et de 1024 octets de mémoire.
3. En anglais, *Event-Triggered Control Paradigm*.

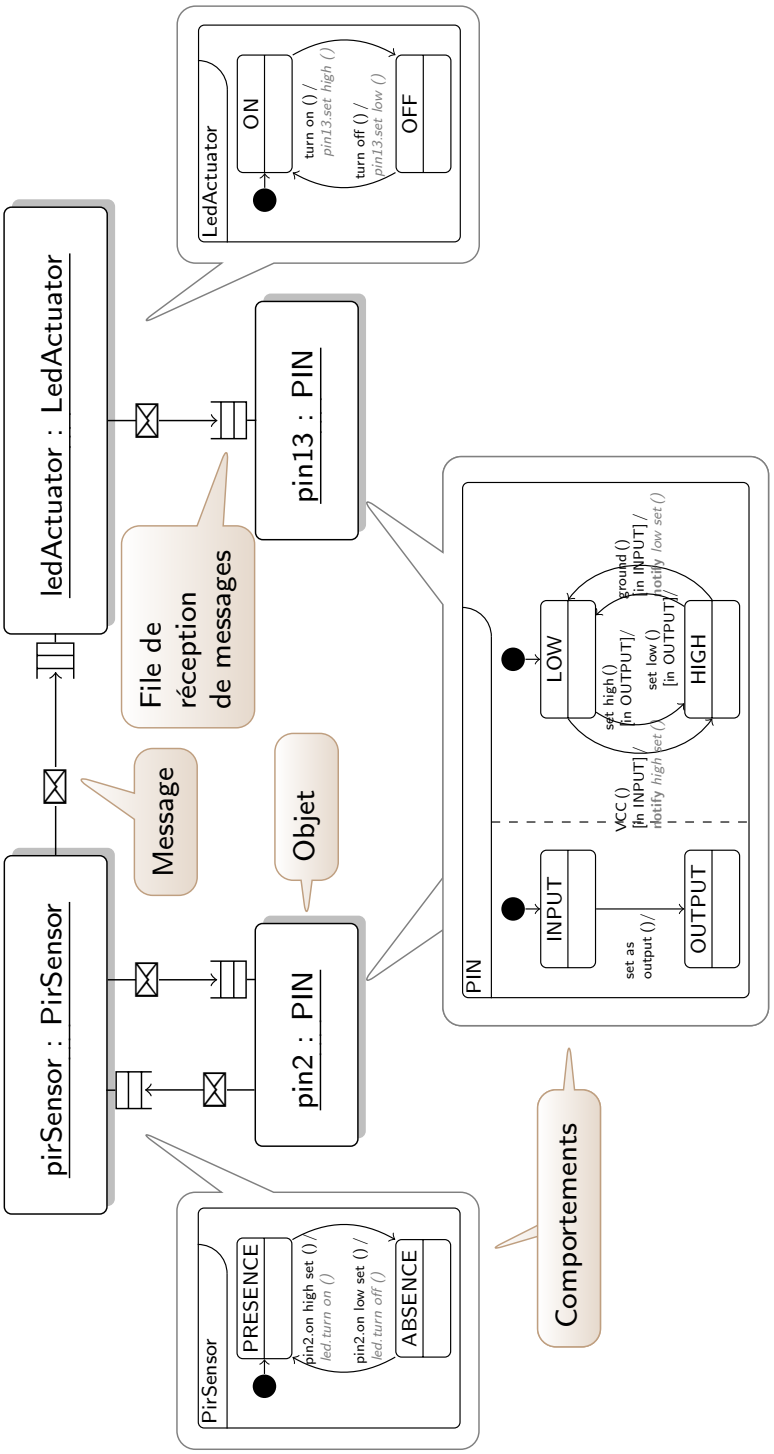


FIGURE 151 – Exemple pris pour la génération de code

Présentation du système. Le système que nous avons choisi de modéliser est un *sous-système* du *système de suivi de passants*, limité à la *détection de présence*. Il est illustré par la figure 151. Son fonctionnement est très simple : Il est constitué de deux objets, un détecteur de présence (*PirSensor*) et une led (*LedActuator*). Lorsque le détecteur détecte une présence, il ordonne à la led de s'allumer (message *turn on ()*). Lorsqu'il détecte une absence, il ordonne à la led de s'éteindre (message *turn off ()*).

Ce système est basé sur la plate-forme Arduino UNO et les objets accèdent à deux broches (*pin*) derrière lesquelles se trouvent des capteurs et actionneurs physiques (un capteur de présence PIR et une led). La broche 2 est configurée en entrée et est connectée au capteur de présence physique. La broche 13 est configurée en sortie et est connectée à la led.

Le paradigme d'envoi de messages. Puisque le langage $\langle \mathsf{HOE} \rangle^2$ se base sur le paradigme d'envoi de messages, il est nécessaire d'identifier les différents éléments impliqués dans ce paradigme. Ces éléments apparaissent dans la figure 151. Chaque objet peut jouer le rôle de *producteur* qui envoie un message à un autre objet qui joue le rôle de *consommateur* du message. Les messages reçus par le consommateur sont stockés dans une *file de réception de messages* en attente de lecture. Un message reçu par un objet est interprété par un *interpréteur* qui va vérifier si le message reçu permet le déclenchement d'une transition de la machine à états de l'objet. Dans le cas d'un déclenchement d'une transition, cet interpréteur peut à son tour envoyer un message vers d'autres objets. Dans le cas où le message ne déclenche aucun franchissement de transition, le message est simplement rejeté.

Afin d'implémenter ce paradigme pour les objets $\langle \mathsf{HOE} \rangle^2$, nous devons donc considérer trois composants pour implémenter le générateur de code : une *file de réception de messages*, la *machine à états* ainsi que son *interpréteur* pour chaque objet du système. Nous présentons dans les trois prochaines parties la génération de code pour ces trois composants. En particulier, nous présentons les éléments à optimiser afin de générer un code optimisé pour la plate-forme, validant la propriété 2 (génération d'un code optimisé).

3.1 Implémentation de la machine à états

La génération de code pour la machine à états est assez triviale. Pour la plate-forme Arduino UNO, nous avons implémenté la machine à états de chaque objet dans le langage Arduino. Cette implémentation nécessite une *énumération* des différents états dans lequel se trouve le système. Dans le cas où la machine à états définirait des régions concurrentes, une énumération est définie pour chacune d'elle.

Le listing 10.2 présente le schéma (*template*) permettant de générer la liste des états des différents objets sous la forme d'énumération. Dans Acceleo, un *template* désigne un schéma de génération de code. Trois requêtes (*query*) sont écrites pour permettre de *normaliser* les chaînes de caractères afin que les noms générés respectent les conventions de nommage des variables, classes, énumération, etc. Le template Acceleo permet, pour chaque région de la machine à états de l'objet, de définir une énumération et un état courant.

```
[comment Used to normalize a name (e.g. region name) /]
[query public normalize(name : String) : String = name.trim().replaceAll(' ','_'
    ) /]

[comment Used to uppercase a normalized name (e.g. message and state names) /]
[query public normalizeUp(name : String) : String = name.normalize().toUpper()
    /]

[comment Used to uppercase first letter of a normalized name (e.g. class name)
    /]
[query public normalizeUpFirst(name : String) : String = name.normalize().
    toUpperFirst() /]


[comment Used to generate state enumerations for each statemachine's region /]
[template public generateEnumerationStates(object : Object) post (trim())]
  [for (region : Region | object.behavior.region)]
  [comment Generate the enumeration /]
  enum State[region.name.normalize()/] {
    [for (state : State | region.subvertex->selectByType(State)) separator (', '
    )][state.name.normalizeUp()/][/for]
  };
  [comment Generate the currentState variable /]
  State[region.name.normalize() /] currentState[region.normalize() /];
  [/for]
[/template]
```

Listing 10.2 – Template de génération des énumérations

```
private:
  enum Stateregion1 {
    INPUT, OUTPUT
  };
  Stateregion1 currentStateregion1;
  enum Stateregion2 {
    HIGH, LOW
  };
  Stateregion2 currentStateregion2;
```

a – Pin.h.

```
private:
  enum Stateregion1 {
    ON, OFF
  };
  Stateregion1 currentStateregion1;
```

b – Led.h.

```
private:
  enum Stateregion1 {
    PRESENCE, ABSENCE
  };
  Stateregion1 currentStateregion1;
```

c – Pir.h.

Listing 3 – Résultat de l'exécution du template Acceleo *generateEnumerationStates*

Le résultat produit par l'exécution du template *generateEnumerationStates* est illustré dans le listing 3. Nous avons généré trois objets : un périphérique de la plate-forme Arduino UNO, une broche (*pin*), et deux objets du *système de détection de présence* hébergés sur le monde Micro-contrôleur de la *plate-forme Arduino UNO*, une led et un capteur de présence PIR. Pour les objets du système hébergés sur la plate-forme, une énumération définit les états de l'objet, et

une variable définit un état courant. Pour la broche de la plate-forme Arduino UNO, la machine à états définit deux régions concurrentes, ainsi que deux énumérations et deux pointeurs sur ces énumérations.

```
[query public initialPseudostate(region : Region) : Pseudostate = self.subvertex
    ->selectByKind(Pseudostate)->any(ps | ps.kind = PseudostateKind::initial) /]
[query public initialState(ps : Pseudostate) : State = ps.outgoing.target.
    oclAsType(State)->asSequence()->first() /]

[template public initializeCurrentStateVariable(object : Object) post (trim())]
[for (region : Region | object.behavior.region)]
  this->current[region.name.normalize()/] = [region.initialPseudostate().
    initialState().name.normalizeUp()/];
  [region.initialPseudostate().doFirstActivity()/]
[/for]
[/template]
```

Listing 10.4 – Template d'initialisation des variables d'état courants

Le listing 10.4 présente un second template permettant d'initialiser la variable pointant sur le premier état des objets. Pour ce faire, ce template cherche l'état initial (*pseudostate* UML) de chaque région de l'objet et franchit la première transition afin de trouver le premier état du système. Deux requêtes supplémentaires sont fournies pour simplifier l'expression du template : *initialPseudostate (region : Region)* permet de trouver l'unique état initial d'une région, tandis que *initialState (ps : Pseudostate)* retourne l'état suivant immédiatement l'état initial. Le listing 5 illustre le résultat produit par cette requête. Tous les états courants sont correctement initialisés, avec à nouveau deux initialisations pour les deux régions de la machine à états de la broche.

```
void Pin::init() {
  this->currentregion1 = INPUT;
  this->currentregion2 = LOW;
}
```

a – Pin.cpp.

```
void LedActuator::init() {
  this->currentregion1 = OFF;
}
```

b – Led.cpp.

```
voir PirSensor::init() {
  this->currentregion1 = ABSENCE;
}
```

c – Pir.cpp.

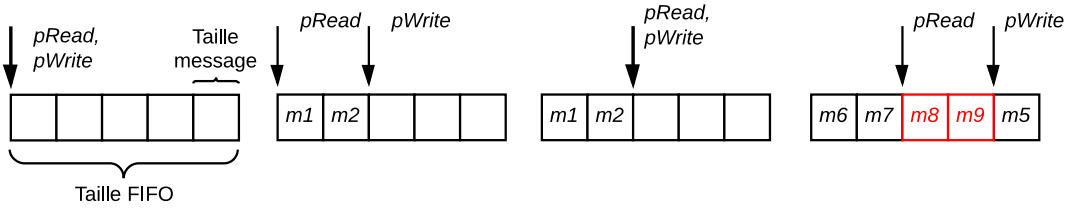
Listing 5 – Résultat de l'exécution du template Acceleo *initializeCurrentStateVariable*

À ce stade, nous avons présenté la génération du code relative à l'implémentation des machines à états des objets, première étape pour concevoir une implémentation du paradigme d'envoi de messages. La prochaine partie présente la génération du code relative au second composant de ce paradigme, la *file de réception de messages*.

3.2 Implémentation de la file de réception de messages

Dans le paradigme d'envoi de messages, la communication entre les objets peut s'effectuer de manière asynchrone. La lecture d'un message ne s'effectue pas forcément immédiatement dès la réception de ce dernier et plusieurs objets peuvent simultanément envoyer des messages à un autre objet sans que ce dernier n'ait le temps de tous les interpréter. Il est donc nécessaire de pouvoir stocker ces messages afin de pouvoir ensuite les lire dans leur ordre d'arrivée. Ces raisons justifient l'implémentation d'une *file de réception de messages*.

Différents travaux ont montré que le correct dimensionnement de la taille de la file permet d'optimiser l'exécution du code sur une plate-forme [Sangiovanni-Vincentelli *et al.* 2000, Obermaisser 2004]. Afin de valider l'hypothèse 2, nous montrons dans cette partie comment nous produisons un code optimisé en influant sur les deux caractéristiques de la file : sa *capacité* (ou *taille*, i.e. le nombre de messages qu'elle peut recevoir) et la *taille d'un message*.



(a) État initial (b) Phase d'écriture (c) Phase de lecture (d) Cas d'erreur (*overrun*)

FIGURE 152 – Implémentation de la file de réception des messages circulaire

Structure et taille de la file. Nous avons implémenté une file circulaire pour la réception des messages dans chaque objet. La file et ses différents états sont illustrés par la figure 152. Elle est caractérisée par différents paramètres (cf. fig. 152a) : sa capacité, la taille des blocs (ici, un bloc correspond à un message), un pointeur de lecture et un pointeur d'écriture. Lors d'une phase d'écriture, le pointeur d'écriture se déplace (cf. fig. 152b) d'autant de blocs que de messages écrits. Lors d'une phase de lecture, l'objet lit (cf. fig. 152c) les messages à partir de l'adresse initiale du pointeur de lecture et chaque lecture provoque le déplacement du pointeur de lecture. Tant que l'adresse du pointeur de lecture est différente de l'adresse du pointeur d'écriture, il reste des messages à lire. La dernière figure 152d illustre un cas possible d'erreur, où sept messages sont écrits avant qu'une nouvelle phase de lecture ne s'active. Dans cet exemple, après avoir écrit les messages *m3*, *m4* et *m5*, le pointeur atteint le bout de la file et revient donc au début de celle-ci. Les messages *m6* et *m7* sont écrits, ce qui positionne le pointeur d'écriture à la même adresse que le pointeur de lecture. Enfin, les messages *m8* et *m9* sont écrits, écrasant les messages *m3* et *m4* qui n'ont pas encore été lus. Ce cas d'erreur illustre la nécessité de dimensionner correctement la file.

Pour optimiser la taille de la file, nous avons fait l'hypothèse que tous les messages écrits durant une phase d'écriture dans la file d'un objet sont lus durant la phase de lecture suivante. Cette hypothèse, qui s'avèrerait fausse sur une architecture asynchrone, est valable sur la plate-forme Arduino UNO, où tous les objets sont exécutés séquentiellement [4]. Nous avons pris pour

4. Le parallélisme est simulé avec la librairie *mthread* [Lamothe 2013], mais la plate-forme elle-même n'offre aucune possibilité matérielle de parallélisme.

seconde hypothèse que la taille de la file répond à l'équation 10.1 :

$$taille_{file} = capacité_{file} * taille_{message_i} \tag{10.1}$$

Où $capacité_{file}$ est le nombre de messages que l'objet peut recevoir dans une phase d'écriture. Il est nécessaire d'optimiser ce nombre afin qu'il soit le plus petit possible de façon à réduire l'empreinte mémoire, tout en étant suffisamment grand pour éviter le cas d'erreur de la figure 152d. La capacité de la file est le premier paramètre d'optimisation que nous ferons par la suite varier pour montrer comment la méthode permet de valider l'hypothèse 2.

Structure et taille d'un message. Afin de réduire la taille d'un message, il est nécessaire de bien le définir sa structure optimale. Nous définissons un message par l'équation 10.2.

$$message = ID_{message} + \sum_{j=0}^{max_{paramètres}} paramètre_j \tag{10.2}$$

La première partie du message contient l'identifiant du message. Cet identifiant est unique pour chaque message qu'un objet peut recevoir. Le message peut contenir un certain nombre de paramètres. $max_{paramètres}$ désigne le nombre maximal de paramètres qui doit être pris en compte. Son calcul est basé sur le message possédant le plus grand nombre de paramètres. Le premier paramètre est *implicite*. Il s'agit d'une référence de l'objet émetteur du message. Cette référence est utilisée dans le cas d'une *réponse* à un message, avec le mot-clé **initiateur**, ou dans le cas d'une notification d'un message, correspondant à l'usage conjoint des deux mots-clés **écouter** et **notifier**. La structure du message étant définie, il est possible de calculer la taille du message à partir de l'équation 10.3.

$$taille_{message} = taille_{ID_{message}} + \sum_{j=0}^{max_{paramètres}} taille_{paramètre_j} \tag{10.3}$$

Où $taille_{ID_{message}}$ est le nombre de bits nécessaire pour encoder l'identifiant du message. Ce nombre est directement calculé à partir du nombre de messages que l'objet peut recevoir aux travers de ces interfaces. Ainsi, si l'objet peut recevoir deux messages, $taille_{ID_{message}}$ vaudra 1 bit, si l'objet peut recevoir entre trois et huit messages, $taille_{ID_{message}}$ vaudra 2 bits, etc.

```
[comment Generate both message structure and messageID enumerations /]
[template public generateMessageStructure(object : Object) post (trim())]
struct message {
  uint8_t messageID : [self.received->size().calulateRequiredBits()/];
  uint16_t initiator : 11;
  [let n : Integer = self.received->getMaximumParameters()]
  [for (i : Integer | Sequence{1..n}) separator ('\n')]uint_8t param[i/] : 6;[/
    for]
  [/let]
};
enum MessageID {
  [for (signal : Signal | object.received) separator (',\n\t')]__HOE2_[signal.
    name.normalizeUp()/]__[/for]
};
[/template]

[comment Generate declarations for one object /]
[template public generateHeaderFile(object : Object) post (trim())]
class [object.name.normalizeUpFirst()/] : public Object {
  public:
    [object.generateMessageStructure()/]
    [comment ... /]
};
[/template]

[comment Generate implementation for one object /]
[template public generateSourceFile(object : Object) post (trim())]
#include "[object.name.normalizeUpFirst().concat('.h')/]"
[object.name.normalizeUpFirst()/]::[object.name.normalizeUpFirst()/] () : Object
    (5, sizeof (message)) {}
    [comment ... /]
[/template]
```

Listing 10.6 – Template de création de la file de réception de messages

La mémoire vive de la plate-forme Arduino UNO est une mémoire SRAM de 2048 octets [5] à adressage direct. Ainsi, nous prenons comme hypothèse que le premier paramètre d'un message, contenant la valeur du pointeur de l'*initiateur* dans la mémoire SRAM est encodé sur onze bits [6]. Nous considérons ensuite comme première approximation que la taille de tous les autres paramètres est fixe et limitée à six bits. Cette approximation nous permet d'encoder un entier non signé entre les valeurs 0 et 63.

Le listing 10.6 fournit trois templates. Le template *generateSourceFile* permet à chaque objet d'hériter du constructeur de la classe *Object* (la déclaration de cette classe est donnée en annexe D) en lui fournissant deux paramètres, le nombre de blocs de la file, et la taille d'un bloc, au moyen de la commande sizeof (message). Le template *generateMessageStructure* permettant la création de la file et des énumérations pour chaque message. Deux autres templates montrent comment sont générés les fichiers *header* et *source* de chaque objet. L'appel du template *generateMessageStructure* est effectué au sein du template *generateHeaderFile*. Il présente la création d'une structure pour chaque objet. Cette structure permet de construire la

5. Pour la version *Atmega328* équipant notre plate-forme.
6. onze bits sont suffisants pour encoder la valeur du pointeur qui ne peut dépasser 2047 – taille maximale de la mémoire vive.

	Avec optimisation (en bits)				Sans optimisation (en bits)			
	messageID	**initiateur**	**paramètres**	**taille**	**messageID**	**initiateur**	**paramètres**	**taille**
Pin	3	11	0	**16**	8	16	0	**32**
Led	1	11	0	**16**	8	16	0	**32**
Pir	1	11	0	**16**	8	16	0	**32**

TABLE 42 – Heuristique d'optimisation de la taille des messages pour réduire l'empreinte mémoire (taille en bits)

définition d'un message. Il est composé d'un identifiant, dont la valeur est calculée, un premier paramètre *initiator* encodé sur 11 bits et une liste de paramètres, chacun encodé sur 6 bits.

Le langage Acceleo (basé sur OCL), ne nous permet pas de faire des fonctions récursives. Pour réaliser la fonction de calcul du nombre de bits requis pour l'encodage de l'identifiant du message, nous avons réalisé un *wrapper* Java présenté dans le listing 10.7.

```
  public Integer calulateRequiredBits(Integer val, Integer power) {
  if (val == 0)
    return 0;
  else if (val <= 2)
    return 1;
  else
    return (((1<<power-1) + 1 <= val) && ((1<<power) >= val)) ? power :
  calulateRequiredBits(val, ++power);
}
```

Listing 10.7 – Calcul du nombre de bits nécessaire à l'encodage d'un entier signé

Ce listing illustre une fonction récursive écrite en Java et appelée depuis Acceleo, permettant de retourner, à partir d'un entier signé, le nombre de bits nécessaire à son encodage. Son appel depuis Acceleo est réalisé avec l'utilisation de la commande *invoke* illustrée par le listing 10.8.

```
[comment Calculate the number of required bits to encore a decimal number /]
[query public calulateRequiredBits(val : Integer) : Integer =
invoke('fr.cea.dacle.canhoe2.faceTracker.services.bitOperation.BitOperation', '
    calulateRequiredBits(java.lang.Integer, java.lang.Integer)', Sequence{val,
    1}) /]
```

Listing 10.8 – Utilisation de la commande *invoke* pour appeler une méthode Java

Le code source généré pour la construction des trois files est illustré par le listing 9. Chaque structure de message est implémentée sous la forme de champs de bits [7]. Ainsi, chaque élément est encodé sur un certain nombre de bits permettant de compresser au mieux la taille des différentes structures. Cela permet de diminuer la taille d'un bloc et ainsi la taille de chaque file de messages. Cette heuristique est particulièrement intéressante pour réduire l'empreinte mémoire de la plate-forme Arduino UNO qui possède une mémoire vive très restreinte (2048 octets) et dont la limite peut être très vite atteinte. La table 42 illustre le gain avec l'application de notre heuristique d'optimisation sur la taille de la structure pour les trois objets. La taille du message est calculée sur la base de la formule de l'équation 10.3. Puisque la structure du message contient un entier signé sur 16 bits, la taille de la mémoire doit être un multiple de

7. En anglais, *bitfields*.

```
class Pin : public Object {
  struct message {
    uint8_t messageID : 3;
    uint16_t initiator : 11;
  };
  enum MessageID {
    __HOE2_SET_AS_OUTPUT__,
    __HOE2_SET_LOW__,
    __HOE2_SET_HIGH__,
    __HOE2_VCC__,
    __HOE2_GROUND__
  };
};
```

a – Pin.cpp.

```
class Pin : public Object {
  struct message {
    uint8_t messageID : 1;
    uint16_t initiator : 11;
  };
  enum MessageID {
    __HOE2_TURN_OFF__,
    __HOE2_TURN_ON__
  };
};
```

b – Led.cpp.

```
class PirSensor : public Object {
  public:
    struct message {
    uint8_t messageID : 1;
    uint16_t initiator : 11;
  };
  enum MessageID {
    __HOE2_HIGH_SET__,
    __HOE2_LOW_SET__
  };
};
```

c – Pir.cpp.

Listing 9 – Résultat de l'exécution du template Acceleo generateMessageStructure

16. Ainsi, si la taille d'un message dans le cas de notre optimisation est de 14 bits, la taille de la structure est dimensionnée sur 16. Sans notre optimisation, la taille du message est de 24 bits, il faudrait donc une structure de 32 bits pour la construire. Cette heuristique qui se veut simple permet de réduire de moitié l'empreinte mémoire de la structure allouée en mémoire. Elle valide donc l'hypothèse 2 en fournissant un code optimisé pour une plate-forme cible.

3.3 Implémentation de l'interpréteur de la machine à états

À ce stade, nous avons présenté l'implémentation de la machine à états et de la file de réception de messages. Avant de pouvoir valider les hypothèses 1 (génération d'un code compilable et sans erreur) et 2 (génération d'un code optimisé), nous avons implémenté le dernier composant du paradigme d'envois de messages : l'*interpréteur de machine à états*. L'interpréteur effectue plusieurs actions :

1. Il récupère et interprète le premier message de la file ;
2. Il vérifie que le message permet de franchir une transition ;
3. Si une transition est franchie, il réalise l'éventuel envoi de messages vers d'autres objets.

Afin d'implémenter l'interpréteur dans chaque machine à états, nous avons conçu une méthode *loop ()* dans chaque objet. Cette méthode s'active tour à tour pour chaque objet par la librairie *mthread*, simulant ainsi une plate-forme asynchrone où chaque objet serait encapsulé

Nombre de LoC générées	
64	lib/LedActuator/LedActuator.cpp
32	lib/LedActuator/LedActuator.h
121	lib/Pin/Pin.cpp
46	lib/Pin/Pin.h
75	lib/PirSensor/PirSensor.cpp
36	lib/PirSensor/PirSensor.h
53	src/sketch.ino
427	total

t

TABLE 43 – Nombre de lignes de code générées

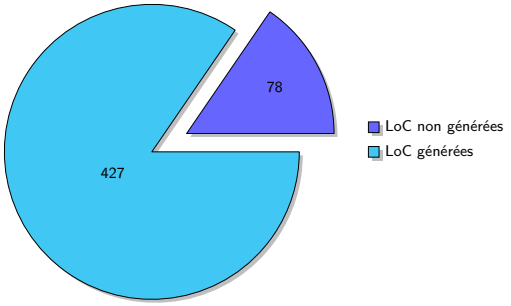


FIGURE 153 – Proportion de code généré

dans un *thread*. Le listing E.1 fourni en annexe E présente le template Acceleo permettant de générer le code de cette méthode et son explication.

L'interpréteur de machines à états est le cœur du paradigme d'envoi de messages. À ce titre, il est nécessaire que ce composant puisse interpréter tout le langage des machines à états et en particulier les différentes méthodes de réception (*réception* ou *écoute*) et d'émission (*diffusion*, *notification*, etc.) de messages. L'écoute et la notification, nécessaires pour la génération du code pour le système de détection de présence sont discutées dans l'annexe F.

3.4 Résultats obtenus et propriétés validées

Cette partie aborde les résultats obtenus pour la génération de code pour l'exemple présenté ci-dessus. Pour valider les propriétés 1 et 4, nous avons regardé de plus près les fichiers générés par l'ensemble des templates Acceleo.

Le tableau 43 présente les fichiers produits ainsi que le nombre de lignes de code générées pour le système présenté. Chaque objet créé possède un fichier de déclaration et un fichier source. Le fichier sketch.ino est un fichier particulier pour la plate-forme Arduino UNO contenant le point d'entrée lu par le *bootloader* de la plate-forme. Au total, 427 lignes de code ont été générées. Le code s'appuie sur différentes librairies, ainsi qu'une classe *Object* (disponible en annexe D) que nous n'avons pas générée. Cette classe représente 78 lignes de code soit 15% du code total de l'application (cf. fig. 153). Nous avons testé la compilation du code, qui s'effectue sans erreur et sans besoin de retoucher le code généré pour fonctionner. En outre, l'exécution du système est bien conforme au comportement attendu. De plus, les objets du système accèdent bien aux périphériques de la plate-forme Arduino UNO, à savoir ses broches. Cela valide les propriétés 1 (*Génération de code compilable et sans erreur*) et 4 (*Accès aux ressources et périphériques de la plate-forme*).

Afin de valider la propriété 2 (*Code optimisé pour la plate-forme*), nous avons étudié l'impact du dimensionnement de la file sur la place occupée par le code dans la mémoire Flash et la place occupée dans la mémoire volatile SRAM durant l'exécuton.

La figure 154 illustre l'impact de l'exécution du code généré sur la mémoire vive de la plate-forme Arduino UNO avec et sans l'heuristique permettant d'optimiser la taille des messages. L'axe des abscisses illustre l'évolution du nombre d'objets générés. L'axe des ordonnés quantifie

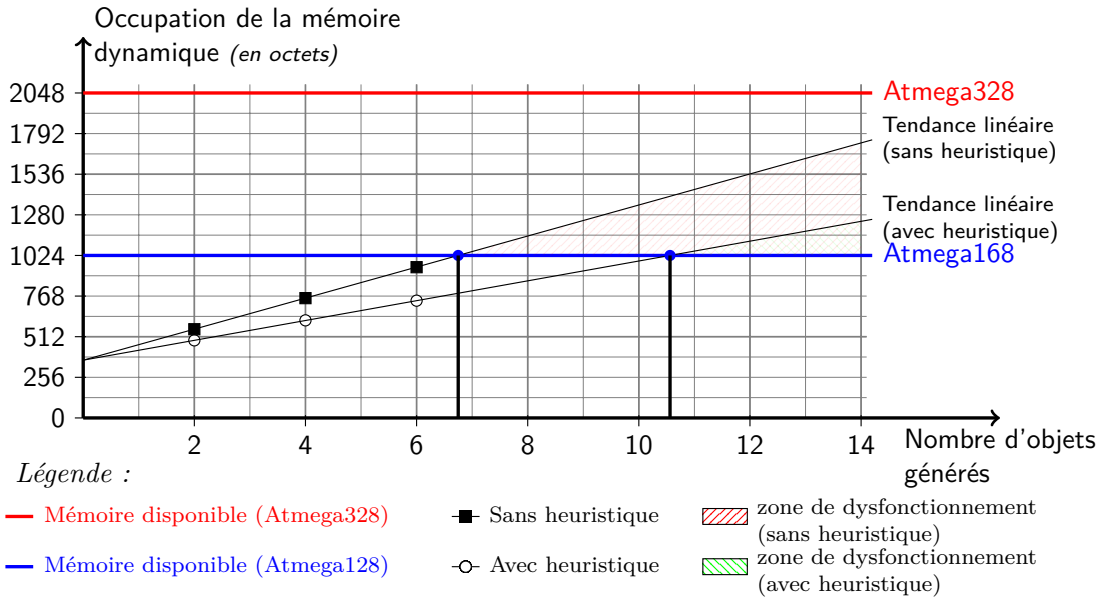


FIGURE 154 – Empreinte mémoire volatile avec et sans heuristique

une taille en octets. Nous avons tracé en trait plein la quantité de mémoire occupée mesurée directement sur la plate-forme. Les valeurs relevées mettent en évidence des courbes de tendance linéaires que nous avons représentées. À titre de comparaison, nous avons également représenté la quantité de mémoire vive disponible pour la plate-forme Arduino UNO pour les processeurs *Atmega168* et *Atmega328*. Nous avons ainsi pu identifier deux points d'intersection entre la tendance réalisée avec et sans heuristique et la mémoire maximale disponible pour le processeur *Atmega128*. Au delà de ces points, des dysfonctionnements peuvent apparaitre dus au manque de mémoire vive pour héberger tous les objets. Pour la tendance avec heuristique, cette limite se situe un peu après les dix objets générés. Pour la tendance sans heuristique, cette limite se situe un peu avant les sept objets générés. Ce graphique valide l'usage de l'heuristique pour satisfaire la propriété 2 (génération de code optimisé).

Nous avons également vérifié que le code généré était suffisamment dimensionné pour être stocké dans la mémoire Flash de la plate-forme Arduino UNO. Les valeurs relevées sont illustrées dans le graphique de la figure 155. Nous pouvons voir que la mémoire occupée par le code de l'application est très en-deçà de la quantité disponible, et l'évolution des objets du système est très peu significative (cela est dû au fait que la plate-forme Arduino UNO charge un certain nombre de librairies qui occupent déjà une partie de la mémoire). On peut également observer que les phases aval peuvent générer des informations en retour (*feedback*) pour le flot des plates-formes.

Tout comme pour la mémoire volatile, nous aurions pu estimer le nombre d'objets à partir duquel la mémoire permanente est insuffisante. Nous nous apercevons néanmoins que la génération d'objets n'influence que très peu la quantité de mémoire occupée et de ce fait n'est pas une limite pour la génération de code. Nous n'avons donc pas eu à fournir d'heuristique pour réduire la place occupée du code en vue de valider la propriété 2. Néanmoins, pour un système plus gros, cet aspect serait également à étudier.

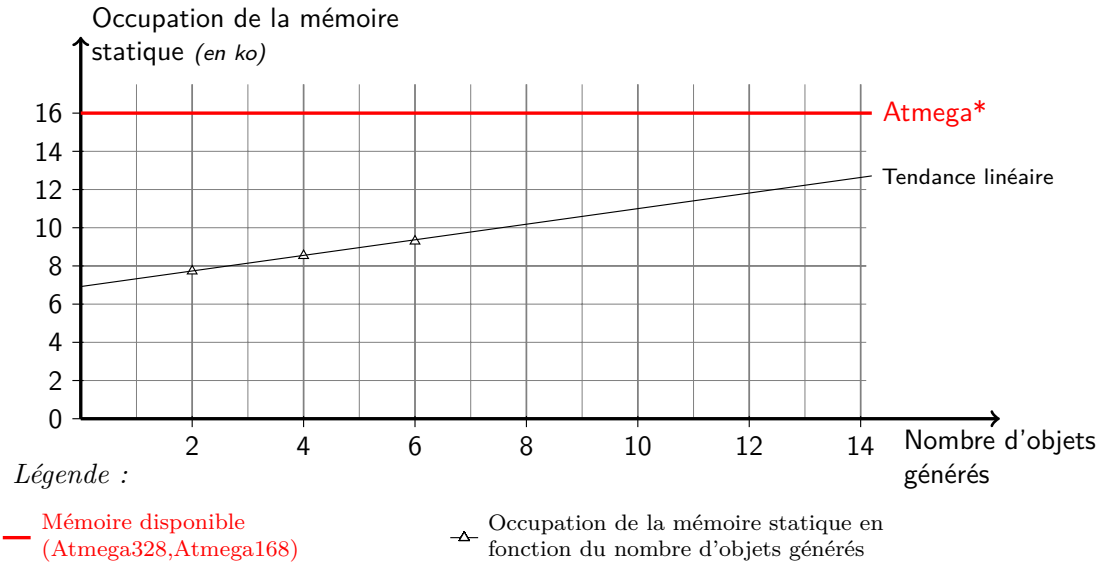


FIGURE 155 – Empreinte mémoire permanente

Cette partie conclut les résultats obtenus sur la génération de code pour la plate-forme Arduino UNO et les hypothèses validées par celle-ci. Les hypothèses validées sont illustrées dans le tableau 44. Nous avons présenté différentes règles de génération de code permettant de générer du code à partir des modèles du système. Les règles de génération sont les mêmes pour la génération d'un code relatif à la plate-forme (par exemple, les broches) et à l'application venant utiliser les services de la plate-forme. Nous pouvons ainsi valider l'hypothèse 5. La génération de code produit un code sans erreur et compilable, avec une proportion de 85% de code généré et 15% écrit à la main. Par ailleurs, il n'est pas nécessaire de modifier le code généré. Cela valide l'hypothèse 1. Le code généré permet l'utilisation des broches de la plate-forme Arduino UNO, validant ainsi l'hypothèse 4.

Nous nous sommes beaucoup intéressés à la génération d'un code optimisé pour la plate-forme en appliquant des heuristiques pour dimensionner correctement la file de réception. Cet apport a permis de réduire l'empreinte mémoire volatile lors de l'exécution de l'application. En réduisant d'une part le nombre de messages que la file peut stocker, et d'autre part la taille d'un message, nous avons pu montrer l'optimisation du code généré. Les mesures relevées dans la figure 154 et les courbes de tendance tracées montrent les optimisations réalisées, validant ainsi

Hypothèses validées	
Hypothèse 1 (Génération de code sans erreur et compilable)	✓
Hypothèse 2 (Code optimisé pour la plate-forme)	✓
Hypothèse 3 (Composition de plates-formes)	
Hypothèse 4 (Accès aux ressources et périphériques de la plate-forme)	✓
Hypothèse 5 (Langage commun pour l'application et la plate-forme)	✓

TABLE 44 – Hypothèses validées par la génération de code sur la plate-forme Arduino UNO

l'hypothèse 2. Le cas présenté est simple et linéaire et nous avons pu produire une estimation pour connaitre la taille mémoire à ne pas dépasser. Ce calcul est basé sur une expérimentation lors de l'exécution du code généré.

Les mesures réalisées quant à l'empreinte mémoire permanente dans la mémoire Flash de la plate-forme Arduino UNO (cf. fig. 155) tendent à prouver que le nombre d'objets générés n'influence que très peu l'espace occupé. Pour cette raison, nous ne nous sommes pas intéressés à l'optimiser du fait de la grande taille (16 ko) de la mémoire. Cette considération est valable pour cette plate-forme et pour la taille des modèles présentés, mais ne le serait plus si l'on considère des modèles plus complexes, ou des architectures plus petites.

En résumé

- ✓ *Illustration de la génération de code pour la plate-forme Arduino UNO*
- ✓ *Intégration d'heuristiques pour optimiser le code généré*
- ✓ *Mesures de l'occupation des mémoires pour valider la génération*

Synthèse du chapitre

Dans ce chapitre, nous avons validé un certain nombre de propriétés à partir du cas d'étude du *système de suivi de passants* présenté, résumées dans la table 45. Ces propriétés ont été étudiées sur les aspects relatifs à la composition de plates-formes et à la génération de code pour des plates-formes spécifiques. Pour la composition de plates-formes, nous avons introduit l'outil EWL que nous avons utilisé pour définir les règles d'implémentation des conteneurs. Pour la génération de code, nous avons présenté l'outil et le langage Acceleo pour la définition des règles de génération de code.

Pour la génération de code, nous nous sommes beaucoup intéressés à la mise en place d'heuristiques permettant de générer un code optimisé pour la plate-forme Arduino UNO. Cette optimisation, ayant permis de valider la propriété 2, s'est basée sur le bon dimensionnement de la file de réception de messages implémentée dans le système. Nous avons illustré cette optimisation par la pratique sans pour autant l'étudier sur le plan théorique.

Ce chapitre clôt l'ensemble de nos contributions. Le chapitre suivant conclut les travaux réalisés durant cette thèse en présentant les perspectives de recherche.

Hypothèses validées	
Hypothèse 1 (Génération de code compilable et sans erreur)	✓
Hypothèse 2 (Code optimisé pour la plate-forme)	✓
Hypothèse 3 (Composition de plates-formes)	✓
Hypothèse 4 (Accès aux ressources et périphériques de la plate-forme)	✓
Hypothèse 5 (Langage commun pour l'application et la plate-forme)	✓

Table 45 – Synthèses des hypothèses validées

Chapitre 11

Conclusion et Perspectives

Liste des sections

Les travaux présentés dans ce mémoire avaient pour enjeux de définir, de formaliser et d'outiller une méthode de développement de systèmes embarqués. Nous avons pour cela établi un pont entre l'ingénierie des systèmes embarqués et celle des méthodes de développement des systèmes d'information. Cela nous a permis d'avoir un regard critique sur les méthodes actuelles issues de la communauté des systèmes embarqués et d'identifier leurs lacunes au niveau des processus, langages et outils de ces méthodes. Nous avons ainsi pu identifier les principales problématiques et formaliser la méthode de développement $\langle \mathsf{HOE} \rangle^2$ pour « *Highly Heterogeneous Object-Oriented Efficient Engineering* » vis-à-vis de ces problématiques. Les quatre principaux résultats de la thèse sont rappelés dans la section suivante.

1 Résumé et rappel des principaux résultats

Afin de répondre à notre problématique qui était la *définition, la formalisation et l'outillage d'une méthode de développement de systèmes embarqués en appliquant les techniques de l'ingénierie des systèmes d'information*, nous avons axé notre contribution autour de quatre résultats principaux : la formalisation d'un processus guidé et d'un langage de modélisation de systèmes embarqués, la composition de plates-formes, l'intégration de la gestion de projet et de la traçabilité couplées aux produits et le développement de l'outil dédié CanHOE2. La table 46 positionne la méthode $\langle \mathsf{HOE} \rangle^2$ par rapport aux différentes caractéristiques que doivent posséder les processus, langages et outils des méthodes de développement de systèmes embarqués et la positionne également vis-à-vis des autres travaux sur les méthodes de développement de systèmes embarqués.

Notre *première contribution* a permis de formaliser le processus $\langle \mathsf{HOE} \rangle^2$ muni de son langage de modélisation en prenant en compte la plupart des spécificités des systèmes embarqués et de leur ingénierie résumées dans le tableau 46. Le processus proposé se distingue par une formalisation claire des rôles et responsabilités de chaque participant du processus, ainsi que la définition des différentes activités, avec leurs états et leurs produits entrants et sortants.

La *seconde contribution* a été la prise en compte du développement de la plate-forme au sein du processus et du langage. Sa prise en compte permet de réduire la complexité de développement des plates-formes en les définissant comment une composition de plates-formes plus simples. Nous avons augmenté la portée de la notion de *plate-forme* par rapport aux autres travaux en l'étudiant du point de vue de l'implémentation de l'application qu'elle héberge et

Processus	⟨HOE⟩²	ACCORD UML	Metropolis	BIP	MOPCOM
Guidage	●	◐	○	●	●
Rigidité	●	○	○	●	◐
Itératif & Incrémental	●	◐	●	◐	○
Traçabilité	●	○	◐	○	○
Réutilisabilité	◐	○	◐	●	○
Rôles & Responsabilités	◐	○	○	○	○
Parallélisme	●	○	○	○	◐
Langage					
Multi-vues	●	●	◐	◐	○
Composabilité	●	◐	◐	●	◐
Support à l'abstraction	●	◐	○	◐	◐
Support à l'hétérogénéité	◐	○	◐	○	●
Degré de formalisme	○	●	◐	●	○
Outils					
Support au processus	◐	◐	◐	●	◐
Support au langage	◐	◐	●	●	○
Support à la gestion de projet	◐	○	○	○	○
Support à la gestion de versions	●	○	○	○	○

TABLE 46 – Synthèse des caractéristiques supportées par la méthode ⟨HOE⟩²

exécute. Nous avons exprimé deux façons de composer une plate-forme et étudié pour chacune d'elle l'impact de la composition sur l'implémentation du système.

La *troisième contribution* a permis d'établir un couplage fort entre les modèles réalisés et les activités de modélisation ayant permis de les obtenir. Cette contribution permet de prendre en compte certaines spécificités de l'ingénierie des systèmes embarqués, telles que le besoin de certification, qui nécessite une traçabilité forte entre les développements réalisés et les besoins qui les justifient, ainsi que la nécessité de mesurer et de piloter la progression du développement. Nous avons posé les premières bases d'une gestion de projet intégrée, pour le moment limitées aux activités d'organisation des équipes et de planification des développements. Ces résultats nous ont permis d'ouvrir des perspectives sur l'intégration d'une gestion de projet plus complète, intégrant d'autres dimensions telles que la gestion des risques, des ressources ou des coûts.

La *quatrième contribution* a concerné le développement de l'outil CanHOE2, un outil dédié à la méthode ⟨HOE⟩² et permettant les supports au langage et au processus par une gestion multi-utilisateurs et multi-rôles. Nous avons également intégré la première brique de gestion de projet concernant l'organisation des équipes et la planification des développements, et nous avons offert un environnement favorable à la gestion de versions dont l'usage s'effectue de manière transparente. Ceci permet de tirer tous les bénéfices des outils de la gestion de versions sans risquer une mauvaise utilisation de ces outils. Nous nous distinguons ainsi des autres travaux

s'appuyant sur des outils génériques qui ne supportent pas ou peu les processus et n'intègrent que très rarement les aspects liés à la gestion de projet ou la gestion des versions des modèles.

2 Perspective de recherche

La figure 156 illustre le travail entrepris et les efforts nécessaires afin de le finaliser. Ces efforts impliquent de poursuivre la formalisation du processus, du langage et le développement de l'outil afin de prendre en compte les différentes caractéristiques partiellement traitées ou non traitées. Du côté du processus, il s'agit d'une part d'identifier les rôles dont nous n'avons pas tenu compte et qui sont exprimés par exemple chez Gérard [Gérard 2000] ou bien chez Sangiovanni-Vincentelli [Sangiovanni-Vincentelli 2007], et d'autre part de favoriser l'identification et la réutilisation au sein du processus de tous les éléments pouvant être réutilisés entre les projets et ne pas se limiter à une simple réutilisation d'une plate-forme d'un projet à un autre. Le langage nécessite un effort supplémentaire de formalisation pour traiter l'hétérogénéité et les différents degrés de formalismes qui ont été insuffisamment abordés durant ces travaux. Enfin, pour l'outillage, il est nécessaire de poursuivre l'effort de développement afin de couvrir les différentes phases de la méthode et de supporter l'ensemble des concepts du langage.

En dehors de la finalisation du travail entrepris, nous avons identifié plusieurs verrous scientifiques et perspectives de recherche qu'il est nécessaire d'aborder en évolution de ces travaux pour prendre en compte toute la complexité des systèmes embarqués.

Perspective 1 (Support au temps-réel) Les travaux que nous avons présentés dans ce mémoire n'abordent pas les aspects temps-réel. Il s'agit cependant d'une caractéristique majeure des systèmes embarqués qui sont en contact direct avec l'environnement physique et qui le mesurent et le pilotent aux moyens de capteurs et d'actionneurs [Kopetz 1997]. La prise en compte des aspects temps-réels est donc nécessaire dans le cadre de la formalisation du langage de la méthode. Certains langages abordent ces aspects. C'est le cas de SPT [1] qui a été intégré dans le langage MARTE [Object Management Group 2011]. MARTE définit par exemple le concept d'horloge et permet de modéliser des systèmes asynchrones où chaque composant possède sa propre base de temps.

Afin de supporter le temps-réel, une perspective à nos travaux de recherche est l'étude des concepts des langages incluant les aspects temporelles et les éventuelles activités qu'il est nécessaire d'intégrer dans le langage et le processus ⟨HOE⟩[2].

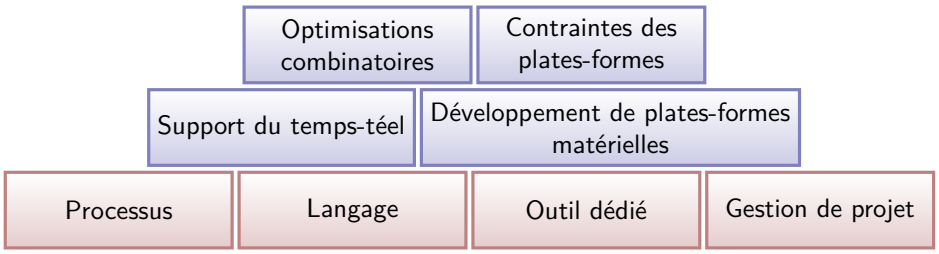


FIGURE 156 – Contributions réalisées (en rouge) et perspectives de recherche (en bleu)

1. En anglais, Schedulabilty Performance and Time (SPT).

Perspective 2 (Prise en compte des contraintes des plates-formes) Un autre aspect non abordé est celui de la prise en compte des contraintes de la plate-forme, telles que la vitesse des processeurs, les temps de latence, d'accès à la mémoire ou le débit dans un bus. Ces contraintes sont nécessaires afin de prendre en compte toute la complexité de développement des systèmes embarqués. Certains langages tels que MARTE permettent d'annoter les modèles avec ces contraintes en vue d'analyses et de simulations.

En continuité de nos travaux, nous pouvons étudier comment les concepts de monde et de conteneur peuvent prendre en compte ces différentes propriétés. Nous avons montré dans le chapitre 10 que certaines considérations pouvaient être prises en compte sur le système. Par exemple, nous avons pu établir une courbe de tendance linéaire montrant l'évolution de la mémoire volatile et permanente occupée en fonction du nombre d'objets du système modélisé. Ces données peuvent être remontées au niveau du développement du système et les différentes contraintes de la plate-forme telles que la taille des mémoires peuvent annoter les ressources et périphériques des modèles des plates-formes. Ces aspects nécessitent d'explorer en profondeur comment les techniques actuelles permettent de prendre en compte toutes ces propriétés.

Perspective 3 (Support au développement des plates-formes matérielles) Dans ces travaux, nous n'avons pas eu le temps d'aborder la génération de plates-formes matérielles. Bien que théoriquement possible par le langage proposé, notre étude de cas n'a pas permis de le montrer. La génération de plates-formes matérielles constitue une perspective intéressante de ces travaux. Pour ce faire, il est intéressant de montrer comment la méthode permettrait de générer au moyen de générateurs spécifiques du code dans des langages de description matérielle tels que VHDL ou VERILOG. Différents travaux abordent ces aspects [Akehurst *et al.* 2007, Wood *et al.* 2008, Vidal *et al.* 2009, Pedroni 2013]. Notamment, les travaux de Akehurst et de Wood. abordent la génération de code VHDL à partir de machines à états [Akehurst *et al.* 2007, Wood *et al.* 2008]. Ces travaux ont nécessité la définition d'un méta-modèle de machine à états, ainsi que d'un méta-modèle du langage VHDL, de règles de transformation entre les deux méta-modèles et de règles de génération de code. Ces travaux supportent un certain nombre de structures des machines à états UML [Wood *et al.* 2008].

Le plus grand verrou scientifique de ces approches est le changement de paradigme entre les machines à états asynchrones et les langages de description matérielle synchronisés sur la définition d'horloges. Cette perspective peut être abordée du point de vue de la définition d'un modèle de plate-forme matérielle, de l'injection dans les conteneurs et de la génération de code dans $\langle \mathsf{HOE} \rangle^2$. Cela nécessite d'évaluer l'effort nécessaire afin de prendre en compte toute la complexité du langage $\langle \mathsf{HOE} \rangle^2$, notamment sur les différents types d'envoi et de réception de messages (e.g. écoute et notification, diffusion de messages) que nous avons identifiés pour répondre aux spécificités des systèmes embarqués.

Perspective 4 (Prise en compte des optimisations combinatoires) Nous avons peu abordé les aspects liés à l'optimisation d'un système en vue de son exécution. Ces aspects ont été plus étudiés dans [Llopard *et al.* 2014] où nous avons proposé une syntaxe de machines à états permettant de décrire et d'optimiser des données complexes. Les travaux de Llopard consistent à définir une chaîne de compilation haut niveau permettant de faire le lien entre une modélisation à base de machines à états et d'envois de messages et des opérations atomiques exécutées par une architecture. Ces travaux se basent sur des techniques de compilation adaptées à la sémantique des machines à états $\langle \mathsf{HOE} \rangle^2$, ceci dans la perspective de permettre l'application de techniques d'optimisation combinatoire. Ces optimisations permettent de sim-

plifier un modèle basé sur le paradigme d'envoi de messages en des opérations directement supportées par l'architecture. Ces travaux ont permis d'identifier deux verrous scientifiques. Le premier est l'identification de toutes les contraintes de l'architecture (telles que le nombre de cycles nécessaires par l'unité arithmétique et logique pour réaliser une opération atomique) afin de proposer une grammaire complète pour la modélisation d'une architecture. Le second verrou est le passage d'une syntaxe haut niveau basée sur l'envoi de messages vers des opérations atomiques de l'architecture à différentes granularités, prenant en compte certaines optimisations, telles que la séquentialisation ou la parallélisation des opérations en fonction des ressources disponibles de l'architecture.

Les quatre perspectives étudiées montrent l'étendue du travail à réaliser pour définir, formaliser et outiller une méthode de développement dédiée aux systèmes embarqués. Les travaux réalisés autour de la méthode $\langle \mathsf{HOE} \rangle^2$ répondent partiellement aux besoins d'ingénierie des SE, il reste cependant de nombreuses extensions à prendre en compte.

Troisième partie

Annexes

Liste des annexes

Annexe A

Dictionnaire des concepts du langage ⟨HOE⟩²

Liste des sections

Cette annexe fournit le dictionnaire de l'ensemble des concepts du langage « *produit* » ⟨HOE⟩². Les concepts sont organisés dans des paquetages pour chaque phase de la méthode et dans un paquetage commun Noyau contenant l'ensemble des concepts communs. La présentation des concepts suit l'ordre chronologique d'apparition dans le processus ⟨HOE⟩².

1 Analyse du besoin

Élément	Description
Acteur	Modélise un acteur externe au système. Un acteur peut être *primaire* (par exemple l'*automobiliste* du système *voiture*), c'est-à-dire un acteur pour lequel le système est conçu ou *secondaire* (par exemple le *réparateur* du système *voiture*), résultant de la conception du système. **Changement par rapport à UML** : En plus de son nom (hérité de UML), un acteur possède un attribut *causalité* permettant de définir sa causalité (*primaire* ou *secondaire*) par rapport au système. Il référence directement un certain nombre de cas d'utilisation qu'il peut exécuter. Une note peut également lui être associée pour le décrire.

Table 47 – Méta-modèle du besoin : Description de l'élément Acteur

Élément	Description
Cas d'utilisation	Modélise un cas d'utilisation du système. Un cas d'utilisation peut être *primaire* (par exemple *conduire la voiture*) ou *secondaire* (par exemple *réparer la voiture*). Son comportement est défini par un ensemble de *scénarios*. **Changement par rapport à UML :** En plus de son nom (hérité de UML), un cas d'utilisation possède un attribut *causalité* permettant de définir une causalité (*primaire* ou *secondaire*) par rapport au système. Il référence un ensemble de scénarios décrivant son comportement ainsi que l'*événement déclencheur* commun à tous ses scénarios. Une note peut également lui être associée pour le décrire. À la différence d'UML, un cas d'utilisation est obligatoirement contenu dans le *système considéré*.

Table 48 – Méta-modèle du besoin : Description de l'élément Cas d'utilisation

Élément	Description
Modèle du besoin	Formalise le modèle du besoin. Il contient le *système considéré* et l'ensemble des *acteurs* externes au système. **Changement par rapport à UML :** le *modèle du besoin* spécialise la méta-classe UML *model* en n'autorisant que l'ajout du système avec ses cas d'utilisation, des acteurs et des associations entre les acteurs et les cas d'utilisation, ainsi que les notes attachées à chacun de ces éléments.

Table 49 – Méta-modèle du besoin : Description de l'élément Modèle du besoin

Élément	Description
Note	Annote un élément du modèle du besoin. **Changement par rapport à UML :** Une note dans le modèle du besoin permet de décrire les différents concepts modélisés (système, acteurs, cas d'utilisation). Dans le cas des *scénarios*, les notes servent à décrire l'état initial du système avant l'exécution, les états intermédiaires et l'état final du système ou de l'acteur.

Table 50 – Méta-modèle du besoin : Description de l'élément Note

Élément	Description
Scénario	Modélise un scénario possible lors de l'exécution d'un *cas d'utilisation* du *système considéré* par un *acteur*. Le scénario peut être *nominal*, traduisant une exécution normale du cas d'utilisation ou d'*erreur*, traduisant une exécution anormale du cas d'utilisation. **Changement par rapport à UML** : Par rapport à la méta-classe UML *Interaction*, un scénario possède un attribut *nature* décrivant sa nature (*nominal* ou d'*erreur*) par rapport au scénario qu'il décrit. Un scénario est inclus dans un cas d'utilisation exécutable par un acteur. Il est constitué de deux lignes de vie, la première *représentant* l'*acteur* et la seconde le *système*. Tous les messages du scénario contiennent un *signal* comme *signature* (en UML, on parle de message *asynchrone*). Le *signal* du premier message se nomme l'*événement déclencheur* et est commun à tous les scénarios d'un même *cas d'utilisation*. Les autres attributs provenant d'UML (le *status* du message par exemple) ne sont pas repris dans $\langle \mathsf{HOE} \rangle^2$.

Table 51 – Méta-modèle du besoin : Description de l'élément Scénario

Élément	Description
Système considéré	Le système à concevoir. Dans le modèle du besoin, il exprime un groupe de fonctionnalités exprimées par des cas d'utilisation. Il est identifié par un nom. **Changement par rapport à UML** : Le système considéré étend la méta-classe *Classifier* du paquetage *Usecases* d'UML. Cette méta-classe ajoute la capacité à un *classifier* UML de posséder des cas d'utilisation.

Table 52 – Méta-modèle du besoin : Description de l'élément Système considéré

2 Noyau

Le paquetage Noyau fournit des concepts utilisés dans les trois dernières phases du processus ⟨HOE⟩². Ils sont répartis dans deux sous-paquetages Object et Comportement.

Objet.

Ce paquetage contient les éléments fondamentaux permettant de concevoir les diagrammes structurels du langage ⟨HOE⟩². Les concepts présents dans ce paquetage étendent le concept *Component* de UML.

Élément	Description
Acteur	Méta-classe permettant la modélisation d'acteurs ⟨HOE⟩². Il étend le concept d'objet et peut à ce titre être associé à n'importe quel objet. **Changement par rapport à UML :** Le concept d'acteur n'est pas utilisé dans les modèles structurels UML. Dans notre cas, il nous semble important de conserver l'acteur puisqu'il permet d'assurer la cohérence des modèles des différentes phases vis-à-vis de la phase d'analyse du besoin.

Table 53 – Méta-modèle des objets : Description de l'élément Acteur

Élément	Description
Objet	Méta-classe permettant la modélisation de tous les composants dans ⟨HOE⟩² pour la conception de systèmes et de plates-formes. Le concept principal dans le langage ⟨HOE⟩² est celui d'*objet*. Il est constitué d'un ensemble des messages qu'il peut émettre vers ou recevoir de la part d'autres objets via des *associations*. **Changement par rapport à UML :** Un objet ⟨HOE⟩² étend la méta-classe UML *Component* afin de pouvoir définir des interfaces *requises* et *fournies*. Par rapport au composant UML, deux associations permettent de lister directement l'ensemble de messages (signaux UML) reçus et émis. En outre, la définition de l'objet restreint celui du composant UML en le contraignant à la définition de son comportement à l'aide d'une *machine à états* UML.

Table 54 – Méta-modèle des Objets : Description de l'élément Objet

Comportement.

Ce paquetage contient les éléments essentiels à la modélisation du comportement des objets du langage ⟨HOE⟩². Il étend essentiellement les concepts des paquetages Action, Communication, permettant la modélisation dynamique dans UML. Cette extension nous permet de modéliser les machines à états selon ⟨HOE⟩².

Élément	Description
Diffusion	Méta-classe permettant la la diffusion d'un message, dans le cadre d'une *association indexée*. **Changement par rapport à UML :** Ce concept étend la méta-classe UML *BroadcastSignalAction*. Dans UML, il n'est toutefois pas indiqué par quelle association le message est retransmis. Nous avons modifié sa sémantique afin d'expliciter la diffusion d'un message à un ensemble d'objets identifié par un nom de rôle d'une association.

Table 55 – Méta-modèle de comportement : Description de l'élément Diffusion

Élément	Description
Écoute	Méta-classe permettant la modélisation de écoute d'un message particulier émis par un objet enregistré en tant que *notifieur* du message. **Changement par rapport à UML :** Ce concept n'existe pas dans UML. Il s'agit de la partie *écouteur* du patron de conception du même nom selon Gamma et al. [Gamma *et al.* 1994]

Table 56 – Méta-modèle de comportement : Description de l'élément Écoute

Élément	Description
Indéxé	Méta-classe permettant la modélisation d'un envoi de message, dans le cadre d'une *association indexée*. **Changement par rapport à UML :** Ce concept étend la méta-classe UML *SendSignalAction* en ajoutant un attribut *index*.

Table 57 – Méta-modèle de comportement : Description de l'élément Indexé

Élément	Description
Notification	Méta-classe permettant la modélisation d'une notification d'un message à tous les objets enregistré en tant qu'*écouteur* du message. **Changement par rapport à UML :** Ce concept n'existe pas dans UML. Il s'agit de la partie *notifieur* du patron de conception *écouteur* selon Gamma et al. [Gamma *et al.* 1994]

Table 58 – Méta-modèle de comportement : Description de l'élément Notification

3 Analyse du système

Le langage d'analyse du système propose des concepts généraux pour la modélisation de systèmes et des concepts spécifiques pour la modélisation de plates-formes.

Élément	Description
Réponse	Méta-classe permettant la modélisation d'une réponse à un signal reçu. **Changement par rapport à UML :** Dans UML, le concept de réponse n'est utilisé que dans le cas d'opération synchrone avec valeur de retour. Dans le cas des systèmes embarqués, nous n'avons considéré que les échanges asynchrones. Ce type d'interaction est décrit dans [Gérard 2000].

TABLE 59 – Méta-modèle de comportement : Description de l'élément Réponse

Système.

Système	Description
Système	Modélise le *système*. Le système est le premier objet du modèle. **Changement par rapport à UML :** Ce concept étend notre concept d'objet. Initialement, il ne possède pas de comportement décrit par une machine à états. L'ensemble des scénarios du besoin constituent des traces partielles légitimes de son comportement.

TABLE 60 – Méta-modèle d'analyse du système : Description de l'élément Système

Plate-forme.

Élément	Description
Conteneur	Objet particulier de la plate-forme servant de réceptacle pour assurer l'implémentation d'objet du système. Il fournit une ou plusieurs règles d'implémentation aux objets qui seront dans les phases suivantes injectés dedans. Il est attaché aux ressources et périphériques de la plate-forme. **Changement par rapport à UML :** Ce concept n'existe pas non plus dans UML et est un autre concept original du langage. Il adresse les problématiques d'implémentation sur une plate-forme d'un objet jusqu'alors indépendant de son implémentation réelle.

TABLE 61 – Méta-modèle d'analyse du système : Description de l'élément Conteneur

Élément	Description
Monde	Objet particulier de la plate-forme permettant d'héberger et d'exécuter des objets. Son comportement adresse les problématiques de la plate-formes (mise en vie, exécution, interruption). Il regroupe des ressources permettant d'héberger et d'exécuter des objets. Il contient un certain nombre de *conteneurs* servant à recevoir des objets. **Changement par rapport à UML :** Le concept de monde n'existe pas dans UML. Il est issu du PDSI [Rygaert 2002]. Il s'agit d'un des deux concepts originaux pour adresser la problématique d'hébergement d'objets.

Table 62 – Méta-modèle d'analyse du système : Description de l'élément Monde

« Rôle »	Description
Périphérique	Un *périphérique* est un rôle particulier de la plate-forme. Les périphériques sont fournis par la plate-forme et offrent des moyens de communication entre des objets hébergés dans des mondes différents. **Changement par rapport à UML :** Ce concept n'existe pas dans UML. Il peut être assimilé à des moyens de communication (bus, NoC) afin de permettre la transmission d'information et de données entre des objets de mondes différents.

Table 63 – Méta-modèle d'analyse du système : Description de l'élément Périphérique

Élément	Description
Plate-forme	Modélise le *plate-forme*. La plate-forme est un système destiné à accueillir une application et ses objets. **Changement par rapport à UML :** Ce concept étend notre concept de système. Initialement, il ne possède pas de comportement décrit par une machine à états. L'ensemble des scénarios du besoin constituent les fragments de son comportement.

Table 64 – Méta-modèle d'analyse du système : Description de l'élément Plate-forme

4 Conception du système

Le langage pour la phase de conception du système comprend des concepts communs pour la modélisation de tous les systèmes considérés (application et plate-forme). Le méta-modèle de conception du système étant un raffinement du méta-modèle de l'analyse du système, les deux seuls nouveaux concepts présentés dans ce méta-modèle sont deux associations permettant de lier les concepts de l'analyse du système.

« Rôle »	Description
Ressource	Une *ressource* est un rôle particulier de plate-forme. Les ressources sont fournies par la plate-forme et regroupées au sein d'un monde. Elles permettent de mettre en vie les objets du système implémenté sur la plate-forme. **Changement par rapport à UML :** Ce concept n'existe pas dans UML. Il peut être assimilé à un processeur et sa mémoire, afin d'exécuter des objets et de manipuler des données.

Table 65 – Méta-modèle d'analyse du système : Description de l'élément Ressource

Association	Description
Composition	Cette association permet de lier le système avec sa plate-forme afin d'héberger les objets de l'application sur les différents mondes. **Changement par rapport à UML :** Ce concept n'existe pas dans UML.

Table 66 – Méta-modèle de conception du système : Description de l'association Composition

Association	Description
Hébergement	Définit la relation d'hébergement des objets sur les mondes de la plate-forme. **Changement par rapport à UML :** Ce concept n'existe pas dans UML.

Table 67 – Méta-modèle de conception du système : Description de l'association Hébergement

5 Implémentation du système

La phase d'implémentation du système introduit une dernière association, offrant le dernier niveau de raffinement du système hébergé sur une plate-forme.

Association	Description
Injection	Définit la relation d'injection des objets dans les différents conteneurs de la plate-forme.

Table 68 – Méta-modèle d'implémentation du système : Description de l'association Injection

Annexe B

Contraintes et Règles de cohérence du processus $\langle$HOE$\rangle^2$

Cette annexe liste les différentes règles de cohérence intra-modèle et inter-modèles qui ont été présentées dans le chapitre 6 en langage naturelle. Les règles sont organisés par méta-modèle. Cette annexe ne présente pas les règles de la première phase du processus $\langle$HOE$\rangle^2$, ces dernières ayant été déjà présentées en langage OCL au chapitre 6.

1 Noyau

```
context Objet::message émis derive:
  self.provided.nestedClassifiers->selectByType(Signal).->asSet()

context Objet::message reçu derive:
  self.required.nestedClassifiers->selectByType(Signal).->asSet()

context State::getAllInternalTransitions () : Set (Transition):
  body: self.region.transition->includesAll (self->asSet()->closure(self.region.
    subvertex->selectByType (State).select(isComposite)))->asSet()

context Statemachine::getAllTransitions () : Set (Transition):
  body: self.region.transition->union (self.region.subvertex->selectByType (
    State).select(isComposite)->getAllInternalTransitions())->asSet()

context Statemachine::getAllSentMessages () : Set (Signal):
  body : self.getAllTransitions().effect.OCLasType(Activity).node->selectByType(
    SendSignalAction).signal->union(self.getAllTransitions().effect.OCLasType(
    Activity).node->selectByType(BroadcastSignalAction).signal)->union(self.
    getAllTransitions().effect.OCLasType(Activity).node->selectByType(
    Notification).signal)

context Statemachine::getAllReceivedMessages () : Set (Signal):
  body : self.getAllTransitions().trigger.event->selectByType(SignalEvent).
    signal

context ActivityNode::getOwner () : Objet
  body:  self.activity.owner.oclAsType(Transition).container.
    containingStateMachine().object

context Event::getOwners () : Set(Objet)
  body:  self.trigger.owner->selectByType(Transition).container.
    containingStateMachine().object->asSet()
```

Listing B.1 – attributs dérivés et requêtes OCL

Le listing 24 fournit différentes règles OCL afin de simplifier la définition des règles de cohérence qui sont détaillées dans le tableau 69. Pour cela, noua définissons les règles pour

Id	Description

[1]	Tous les messages intervenant dans un *envoi* de message d'un objet sont forcément des message *émis* par l'objet, c'est-à-dire contenus par l'une de ses interfaces *fournies*.
	`context Objet inv:`
	`  self.message émis.includesAll(self.comportement.getAllSentMessages())`

[2]	Tous les messages intervenant dans une *réception* de message d'un objet sont forcément des messages *reçus* par l'objet, c'est-à-dire contenus par l'une de ses interfaces *requises*.
	`context Objet inv:`
	`  self.message reçu.includesAll (self.comportement.`
	`  getAllReceivedMessages ())`

[3]	Un envoi de message d'un objet *expéditeur* vers un objet *destinataire* n'est valide que si le *destinataire* est directement accessible par l'*expéditeur* au travers d'une association.
	`context ActivityNode::getOwner () : Object`
	`body:  self.activity.owner.oclAsType(Transition).container.`
	`  containingStateMachine().object`
	
	`context SendSignalAction inv:`
	`  self.getOwner().nestedClassifier->selectByType(Property)->one((type =`
	`  self.target.type).and(name = self.target.name))`
	
	`context Diffusion inv:`
	`  pre: self.association.type.oclIsTypeOf(Association)`
	`  self.getOwner().nestedClassifier->selectByType(Association)->one(name`
	`  = self.association.name)`

[4]	Une *diffusion* de message ou un envoi de message *indexé* implique que le message est transmis sur une association *multi-valuée* et *ordonnée*.
	`context Diffusion inv:`
	`  let a : Association = self.association.type.oclAsType(Association)`
	`  in a.memberEnd->excluding(ownedEnd)->select (p : Property | p.`
	`  isMultiValued().and(p.isOrdered())`
	
	`context Indexé inv:`
	`  self.target.isMultiValued().and(self.target.isOrdered)`

[5]	Si un objet (l'*écouteur*) *écoute* le message émis par un second objet (le *notifieur*), alors le *notifieur* doit être accessible au travers d'une association et le message doit appartenir à une interface fournie par le *notifieur* et requise par l'*écouteur*.
	`context Écoute inv:`
	`  self.getOwners()->any(Objet).nestedClassifier->selectByType(Property)`
	`  ->one((type = self.notifieur.type).and(name = self.notifieur.name))`

TABLE 69 – Liste des règles de cohérence intra-modèle du noyau

les deux attributs dérivés *message émis* et *message reçu* du concept *Objet* du méta-modèle ⟨HOE⟩² (cf. fig. 46). Nous définissons également cinq requêtes OCL. `getAllInternalTransitions()` et `getAllTransitions()` permettent de lister l'ensemble des transitions d'une machine à états. `getAllSentMessages()` et `getAllReceivedMessages()` permettant de lister tous les signaux reçus et émis intervenant dans la modélisation d'une machine à états (i.e. dans la modélisation du

comportement d'un objet ⟨HOE⟩[2]). Les deux requêtes `getOwner()` permettent, à partir du concept *ActivityNode* ou à partir du concept *SignalEvent* d'UML de récupérer la machine à états dans laquelle cet envoi ou réception de signal est impliqué. Ces deux dernières requêtes utilisent la requête `containingStateMachine()` définie par UML [OMG 2010, p. 558]. Cette requête (récursive) permet, à partir d'une région (pouvant être une région d'une machine à états ou d'un état composite), de retrouver la machine à états qui la contient.

Le tableau 69 détaille les règles de cohérence au sein du méta-modèle du *noyau*. Ces règles ont été expliquées dans le chapitre 6. La contrainte quatre utilise la requête particulière `isMultiValued()`, définie par UML [OMG 2010, p. 96]. Cette requête permet de vérifier si l'association est bien multi-valuée.

2 Analyse du système

Id	Description

[1]	Les acteurs du système ne peuvent communiquer qu'avec le système et non un objet composant le système .
	`context Acteur inv:`
	`  self.nestedClassifier->selectByType(Property)->forAll(p : Property | p.type.oclIsTypeOf(System))`

TABLE 70 – Règle de cohérence intra-modèle pour l'analyse d'une application

Id	Description

[2]	Un objet de la plate-forme joue le rôle de *périphérique* lorsqu'il est directement contenu par la plate-forme.
	`context Plate-forme inv:`
	`  self.périphérique.includesAll(self.nestedClassifier->selectByType(Property)->select (p : Property | p.type.oclIsTypeOf(Object)))`

[3]	Un objet de la plate-forme joue le rôle de *ressource* lorsqu'il est directement contenu par un monde de la plate-forme.
	`context Monde inv:`
	`  self.ressource.includesAll(self.nestedClassifier->selectByType(Property)->select (p : Property | p.type.oclIsTypeOf(Object)))`

[4]	Un conteneur ne peut être associé qu'aux périphériques de la plate-forme et aux ressources du monde auquel il appartient.
	`context Conteneur inv:`
	`  self.owner.oclAsType(Monde).ressource->includesAll(self.ressource)`
	
	`context Conteneur inv:`
	`  self.owner.owner.oclAsType(Plate-forme).périphérique->includesAll(self.périphérique)`

TABLE 71 – Liste des règles de cohérence intra-modèle pour l'analyse d'une plate-forme

Les tableaux 70 et 71 détaillent les contraintes OCL exprimées pour les règles de cohérence intra-modèle et inter-modèles pour la phase d'*analyse du système*.

3 Conception du système

Id	Description

[1]	Les objets contenus dans le modèle de conception sont obligatoirement hébergés dans les mondes de la plate-forme référencée par le système en conception.
	`context System inv:`
	`  self.object.monde->notEmpty()`

[2]	Si un objet est découpé en plusieurs fragments durant l'activité de distribution, chaque fragment doit porter le même nom que l'objet initial.
	`context Découper un objet d'analyse inv:`
	`  self.fragments d'objet->forAll(o : Objet | o.name = self.objet à découper.name)`

[3]	Les fragments d'un même objet doivent être répartis dans des mondes différents.
	`context Distribuer les fragments d'objet dans les mondes inv:`
	`  self.fragments d'objet->forAll(o1, o2 : Object | o1.monde <> o2.monde)`

[4]	Deux ou plusieurs fragments d'un objet hébergés dans des mondes différents doivent pouvoir communiquer entre eux aux travers d'associations.
	`context Plate-forme inv:`
	`  self.monde.objet->forAll(o1, o2 : Objet | o1.name = o2.name implies o1.monde <> o2.monde)`

TABLE 72 – Liste des règles de cohérence intra-modèle pour la conception du système

Les tableaux 72 et 73 listent les règles de cohérence intra-modèle et inter-modèles pour la conception du système. Dans ces règles, nous voyons pour la première fois l'expression d'une contrainte appliquée sur le concept de *Tâche* que nous avons défini dans ⟨HOE⟩² (cf. chapitre 8). Ceci est exprimé au moyen des contextes OCL `context nom de la tache inv: .`

Cette syntaxe est particulière puisque d'une part, nous changeons de niveau de modélisation, et d'autre part, cette syntaxe n'est pas permise dans UML. La contrainte exprimée s'applique au niveau de modélisation *M1* (modèle) selon [Rolland 1993] et non au niveau *M2* (méta-modèle) telle que le sont les autres règles exprimées dans cette partie et dans cette thèse. Une règle OCL est possible au niveau *M1* dès le moment où le concept peut être instanciée. La contrainte s'applique alors aux instances et vérifient leurs conformités par rapport au modèle.

Dans le cas des diagrammes d'activités UML, il n'est normalement pas possible d'appliquer des contraintes aux produits entrants et sortants des nœuds d'activités, puisqu'il n'est pas possible d'instancier le concept de *Pin*. Nous nous le permettons dans ⟨HOE⟩² puisque le concept de *Tâche* du langage spécialise à la fois le concept de *Action* et le concept de *Classifier* (cf. fig. 97). À ce titre, la tâche est instanciable et leurs instances (les *feuilles de tâche*) peuvent donc être testées par un interpréteur OCL. Ces contraintes sont pour le moment théorique et nécessitent de tester le support des outillages afin de vérifier leurs interprétabilités.

Id	Description

[1]	À un objet d'analyse correspond un ou plusieurs objets de conception (fragments) dans le modèle de conception du système. Les fragments d'objet portent le même nom que l'objet d'analyse.
	`context Découper un objet d'analyse inv:`
	`  self.fragments d'objet->one (o : Objet | o.monde->notEmpty().and(o.`
	`  name = self.objet à découper.name))`

[2]	À une association entre deux objets dans le modèle d'analyse de l'application correspond au moins une association entre des fragments des mêmes objets dans le modèle de conception du système.
	`context Découper un objet d'analyse inv:`
	`  self.objet à découper.nestedClassifier->selectByType(Association)->`
	`  forAll(self.fragments d'association->select (a : Association | a.`
	`  name = self.name)->notEmpty())`

[3]	Un monde du modèle d'analyse de la plate-forme correspond à un monde du *modèle de conception*
	`context Initialiser la phase de conception du système inv:`
	`  self.modèle des plates-formes d'hébergement.système.oclAsType(Plate-`
	`  forme).monde->includesAll(self.monde de conception)`

TABLE 73 – Liste des règles de cohérence inter-modèles pour la conception du système

4 Implémentation du système

Les tableaux 74 et 75 expriment enfin les règles de cohérence intra-modèle et inter-modèles pour la phase d'implémentation du système. Les explications de ces règles sont fournies dans le chapitre 6.

Id	Description

[1]	Les objets hébergés sur un monde de la plate-forme ne peuvent être injectés que dans les conteneurs fournis par ce monde.
	`context Objet inv:`
	`  self.monde.conteneur->includesAll(self.conteneur)`

[2]	Lorsqu'un objet injecté dans un conteneur est transformé par les règles d'implémentation du conteneur, celui-ci remplace ce dernier, toutes les associations issues du conteneurs vers les périphériques et les ressources sont également référencées par l'objet.
	`context Éxécuter les règles d'implémentation inv:`
	`  self.objet injecté.conteneur.périphérique->union(self.objet injecté.`
	`  conteneur.ressource)->forAll(o : Object | self.objet implémenté.`
	`  nestedClassifier->selectByType(Property)->select(p : Property | p.`
	`  type = o)->notEmpty()`

TABLE 74 – Liste des règles de cohérence intra-modèle pour l'implémentation du système

Id	Description

[1]	À un objet du modèle de conception correspond le même objet dans le modèle d'implémentation.
	`context Initialiser la phase d'implémentation du système inv:`
	`  self.système implémenté.objet->includesAll(self.système hébergé.objet)`

[2]	À une association entre deux fragments objets hébergés dans le même monde dans le modèle de conception du système correspond la même association dans le modèle d'implémentation du système.
	`context Initialiser la phase d'implémentation du système inv:`
	`  self.système implémenté.objet.nestedClassifier->selectByType(`
	`  Association)->includesAll(self.système hébergé.objet.`
	`  nestedClassifier->selectByType(Association))`

TABLE 75 – Liste des règles de cohérence inter-modèles pour l'implémentation du système

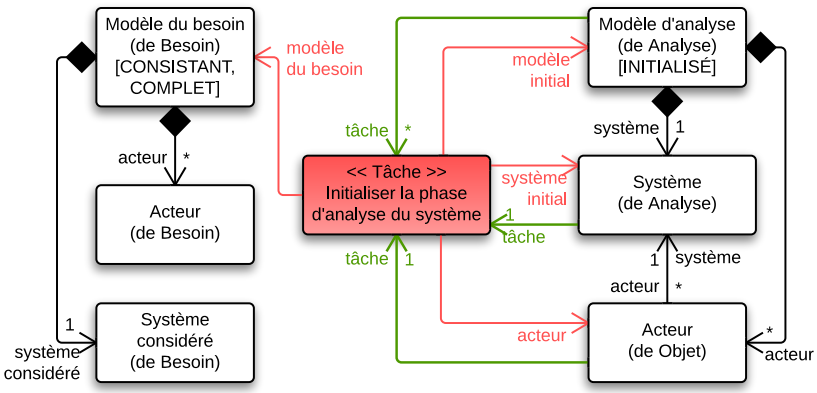
Annexe C

Description des différentes tâches du processus

Liste des sections

Cette annexe liste les différentes tâches du processus, les tâches pour la phase d'analyse du besoin étant fournies dans le chapitre 8 (cf. tab. 32 à 35).

1 Analyse du système

Tâche	Description
Initialiser la phase d'analyse du système	Initialise le modèle du système à partir du modèle d'analyse du besoin. **Réalisation :** chef de projet (automatisable) **Pré-condition :** modèle d'analyse du besoin CONSISTANT ou COMPLET **Post-condition :** modèle d'analyse du système INITIALISÉ
	

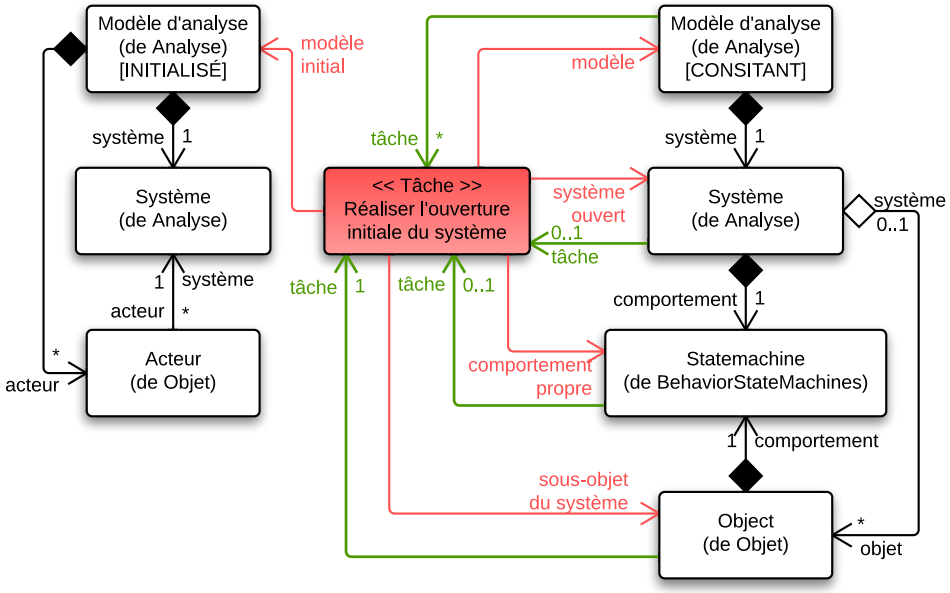
Explication : initialement, le modèle d'analyse du besoin est CONSISTANT ou COMPLET. Cette tâche *initialise* la phase d'*analyse du système*. Le modèle d'*analyse du système* contient un *système* correspondant au *système considéré* du modèle d'*analyse du besoin*, et un *acteur* pour chaque *acteur* du modèle d'*analyse du besoin*. Tous les acteurs référencent le système.

Table 76 – Gestion de projet : Tâche d'intialisation de la phase d'analyse du système

La phase d'analyse du besoin permet d'ouvrir hiérarchiquement une application ou une plate-forme de la même façon. Dans les tableaux 76 à 80, il est donc possible de remplacer *Système* par *Plate-forme*, puisque le concept de *plate-forme* spécialise celui de *système*.

Toutes les tâches de modélisation à partir de la phase d'analyse du système visent à satisfaire les scénarios formalisés durant l'analyse du besoin. La traçabilité au besoin est assurée, selon le méta-modèle des activités et de la gestion de projet (cf. fig. 97 de la page 157), par la *feuille de tâche* qui est une instance du concept de *tâche*. Pour cette raison, les *scénarios* n'apparaissent pas comme produits en entrée des tâches, mais sont liés aux *feuilles de tâche* via les références « *à satisfaire* » et « *satisfait* ».

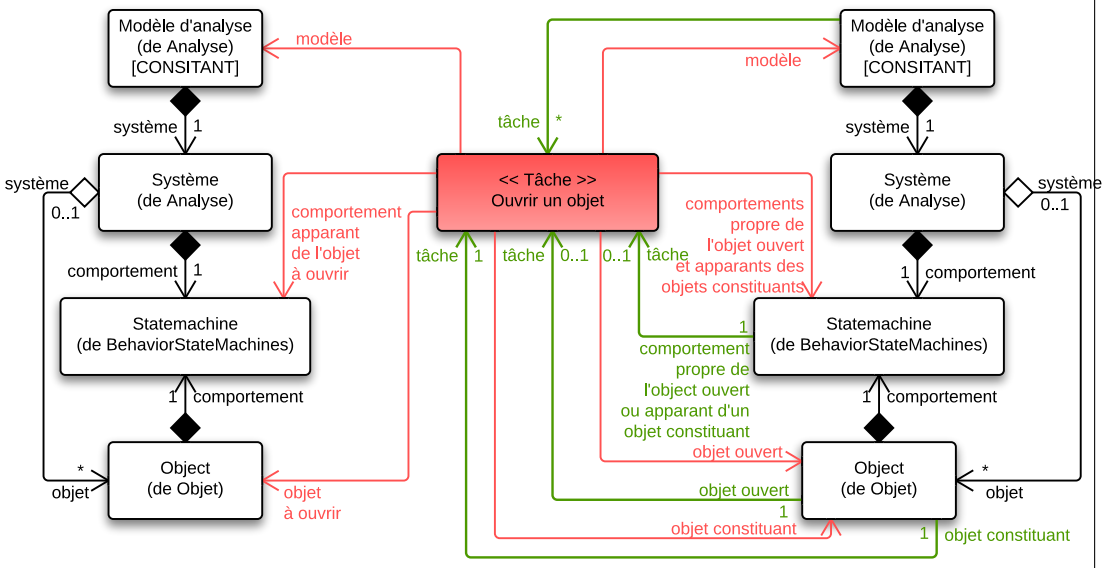
Tâche	**Description**
Réaliser l'ouverture initiale du système	Réalise l'ouverture initiale du système. **Réalisation :** développeur **Pré-condition :** modèle d'analyse du système INITIALISÉ **Post-condition :** modèle d'analyse du système CONSISTANT



Explication : le modèle d'*analyse du système* est INITIALISÉ et contient le système *fermé* dont le comportement n'est pas encore défini sous forme de machine à états. La tâche d'*ouverture initiale du système* consiste à ouvrir le système et détailler ses objets constituants, ainsi que son comportement propre. Le système *fermé* est alors remplacé par son ouverture dans le modèle d'*analyse du système*.

TABLE 77 – Gestion de projet : Tâche d'ouverture initiale du système

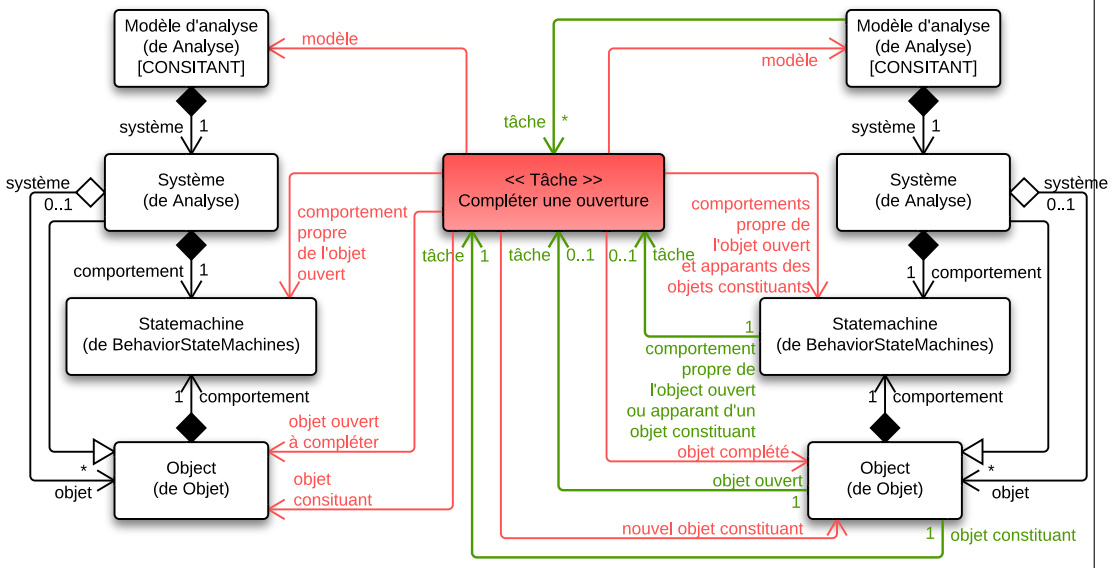
Tâche	Description
Ouvrir un objet	Réaliser l'ouverture d'un objet. **Réalisation :** développeur **Pré-condition :** modèle d'analyse du système CONSISTANT contenant un objet fermé **Post-condition :** objet ouvert contenant un comportement propre et des objets constituants



Explication : un objet *fermé* du système contient un comportement *apparent*. La tâche d'*ouverture d'un objet* consiste à *ouvrir* cet objet et détailler ses objets constituants, ainsi que son comportement propre. L'objet *fermé* est alors remplacé par son ouverture dans le modèle d'analyse.

TABLE 78 – Gestion de projet : Tâche d'ouverture d'un objet

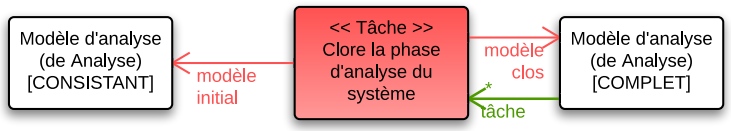
Tâche	Description
Compléter une ouverture	Compléter une ouverture d'un objet. **Réalisation :** développeur **Pré-condition :** modèle d'analyse CONSISTANT contenant un objet déjà ouvert **Post-condition :** un objet ouvert dont l'ouverture est complétée, contenant un comportement propre et des objets constituants



Explication : un objet *ouvert* du système contient un comportement propre. La tâche de *complétion d'ouverture* consiste à *compléter* l'ouverture de cet objet en ajoutant des objets constituants, ainsi qu'en raffinant son comportement.

TABLE 79 – Gestion de projet : Tâche de complétion d'une ouverture

Tâche	Description
Clore la phase d'analyse du système	Clôt la phase d'analyse du système. **Réalisation :** chef de projet (automatisable) **Pré-condition :** modèle d'analyse du besoin COMPLET, modèle d'analyse du système CONSISTANT **Post-condition :** modèle d'analyse du syst !me COMPLET



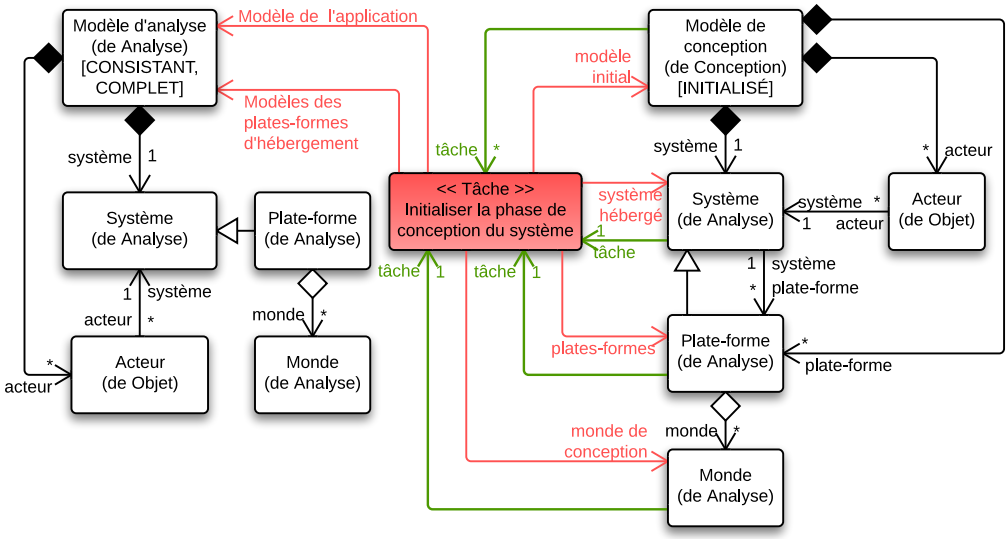
Explication : cette tâche permet de clore la phase d'analyse du système. Cette phase peut être close lorsque la phase d'analyse du besoin est-elle même close et que toutes les tâches d'analyse du système ont permis de satisfaire tous les scénarios du modèle d'analyse du besoin en phase d'analyse du système. La tâche de clôture de la phase d'analyse du système permet de *compléter* le modèle d'analyse du système qui devient COMPLET.

TABLE 80 – Gestion de projet : Tâche de clôture de la phase d'analyse du système

2 Conception du système

Tout comme durant la phase d'analyse du système, les quatre tâches de la phases de conception du système permettent à la fois la modélisation de la partie applicative d'un système que le développement d'une plate-forme, selon la composition de plate-forme détaillée dans le chapitre 7. Les tableaux 81 à 84 détaillent les quatre tâches de la phase de conception du système.

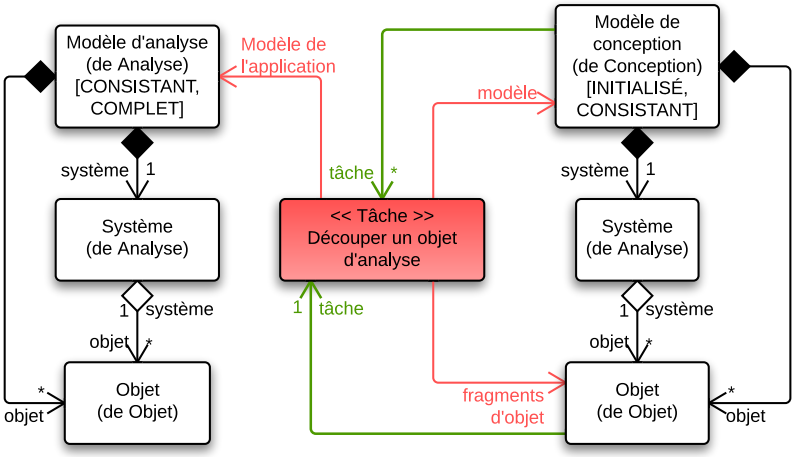
Tâche	Description
Initialiser la phase de conception du système	Initialise la phase de conception du système et le modèle de conception du système à partir des modèles d'analyse de l'application et des plates-formes. **Réalisation :** chef de projet (automatisable) **Pré-condition :** modèles d'analyse de l'application et des plates-formes CONSISTANTS ou COMPLETS **Post-condition :** modèle de conception du système INITIALISÉ



Explication : initialement, les modèles d'*analyse de l'application et des plates-formes* hébergeant l'application sont CONSISTANTS ou COMPLETS. Cette tâche *initialise* la phase de *conception du système* avec le modèle de *conception du système*. Il contient le *système* du modèle d'*analyse de l'application* et les *plates-formes* des modèles d'*analyse des plates-formes*, et établit le lien de *composition* entre le système et les plates-formes. Le modèle reprend également les *acteurs* du *modèle d'analyse de l'application*. Dans l'état initial, les objets du modèle d'*analyse de l'application* ne sont pas encore importés dans le modèle de conception du système.

TABLE 81 – Gestion de projet : Tâche d'initialisation de la phase de conception du système

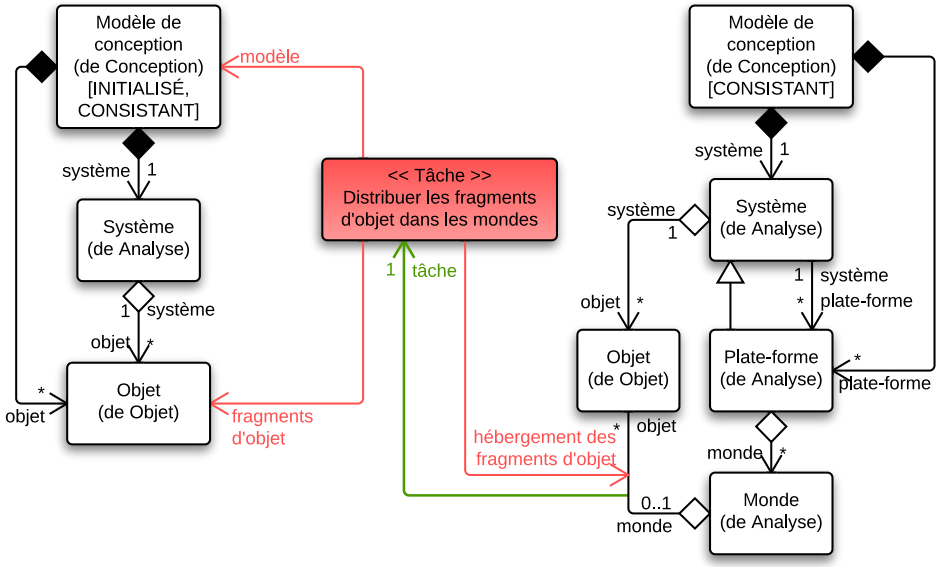
Tâche	Description
Découper un objet d'analyse	Découpe un objet du modèle d'*analyse du système* en fragments d'objet en vu de leur distribution dans les différents mondes de la ou les plates-formes hébergeant l'application. **Réalisation :** développeur **Pré-condition :** modèle de conception du système INITIALISÉ ou CONSISTANT **Post-condition :** l'objet a été découpé en fragment



Explication : la tâche de *découpage d'un objet d'analyse* consiste à prendre un objet du modèle d'*analyse du système* et à le *découper* en plusieurs *fragments* d'objet en vue de leur distribution dans les différents mondes de la ou les plates-formes hébergeant la partie applicative du système. Pour réaliser cette tâche, le modèle de conception du système peut être dans l'état INITIALISÉ ou CONSISTANT. À l'issue de cette tâche, le modèle de *conception du système* contient les fragments d'objet qui ne sont pas encore hébergés dans les mondes de la ou les plates-formes. Cette tâche constituant la première partie de l'activité de *distribution* (l'autre partie étant la *distribution* des fragments sur les mondes, selon la figure 68), elle n'altère pas l'état du modèle de *conception du système*.

TABLE 82 – Gestion de projet : Tâche de découpage d'un objet d'analyse

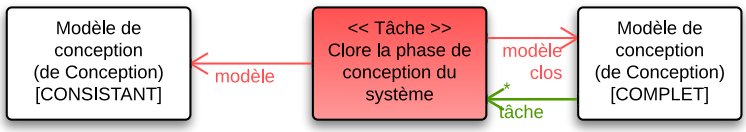
Tâche	Description
Distribuer les fragments d'objet dans les mondes	Distribue les fragments d'objets obtenus durant la tâche de *découpage d'un objet d'analyse* dans les différents mondes de la ou les plates-formes hébergeant la partie applicative du système **Réalisation :** développeur **Pré-condition :** modèle de conception du système INITIALISÉ ou CONSISTANT **Post-condition :** modèle de conception du système CONSISTANT



Explication : les fragments d'objet issus de la tâche de *découpage d'un objet d'analyse* sont distribués durant cette tâche dans les différents mondes de la ou les plates-formes hébergeant la partie applicative du système. À l'issue de cette tâche, au moins un objet ou des fragments d'objet sont distribués dans les mondes. Le modèle de *conception du système* devient CONSISTANT.

TABLE 83 – Gestion de projet : Tâche de distribution des fragments d'objet dans les mondes

Tâche	Description
Clore la phase de conception du système	Clôt la phase de conception du système. **Réalisation :** chef de projet (automatisable) **Pré-condition :** modèles d'analyse de l'application et des plates-formes COMPLET, modèle de conception du système INITIALISÉ ou CONSISTANT **Post-condition :** modèle de conception du système COMPLET



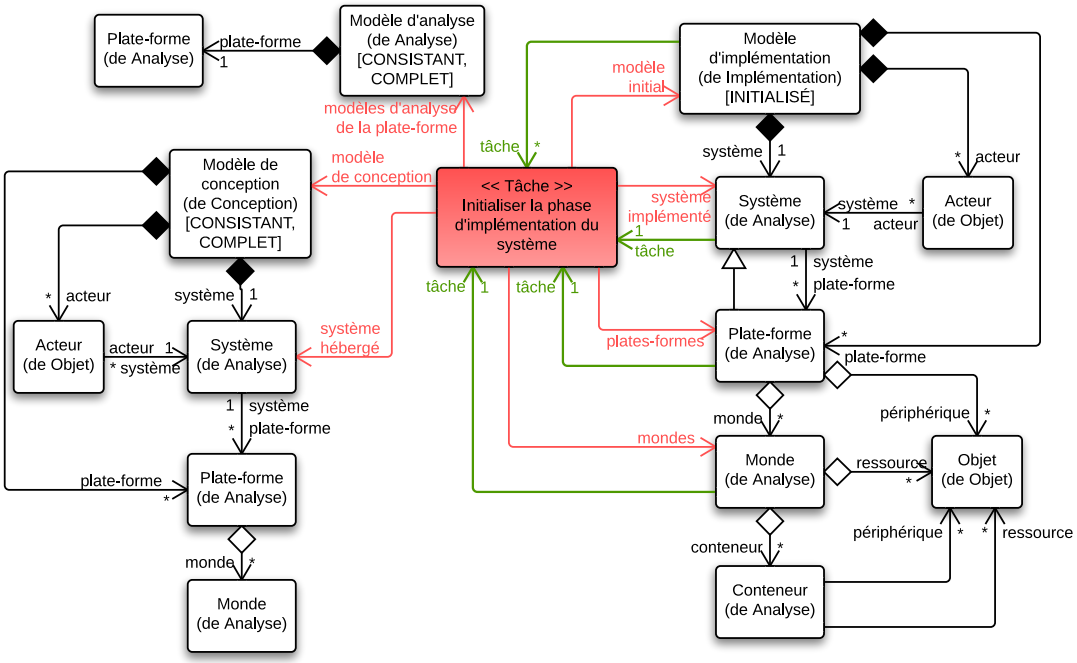
Explication : cette tâche permet de clore la phase de conception du système. Cette phase peut être close à condition que la phase d'analyse du système pour le développement de la partie applicative du système et toutes ses plates-formes est elle-même close et que toutes les tâches de conception du système ont permis de satisfaire tous les scénarios du modèle d'analyse du besoin en phase de conception du système. La tâche de clôture de la phase de conception du système permet de compléter le modèle de conception du système qui devient COMPLET.

TABLE 84 – Gestion de projet : Tâche de clôture de la phase de conception du système

3 Implémentation du système

Les tableaux 85 à 88 détaillent les quatre tâches durant la phase d'implémentation du système.

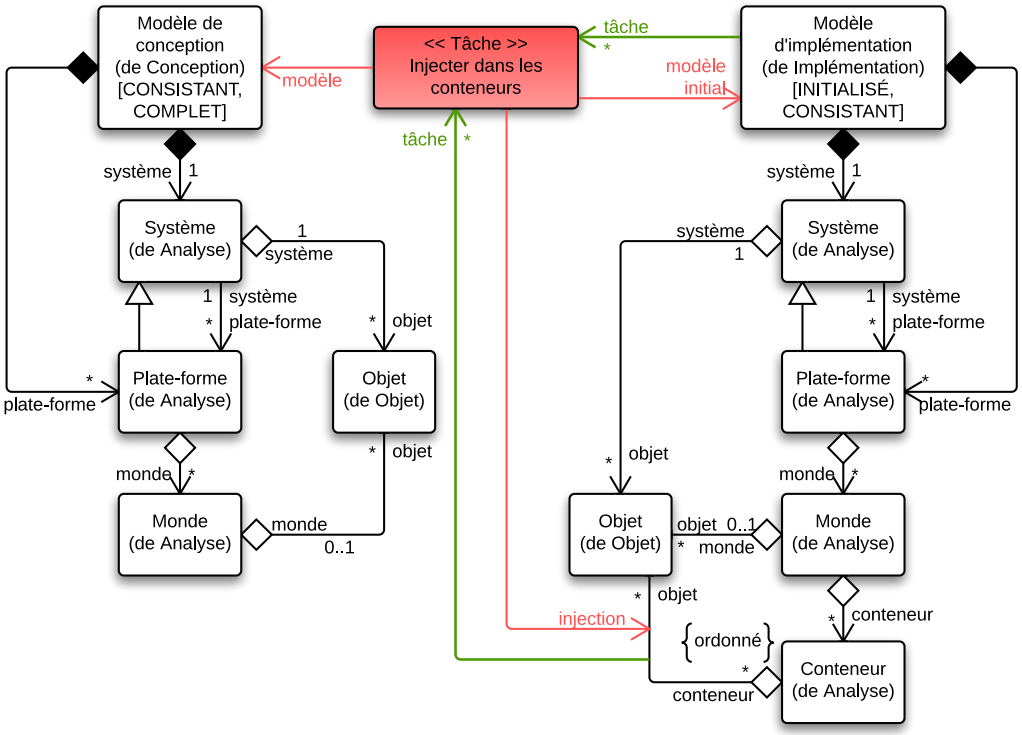
Tâche	Description
Initialiser la phase d'implémentation du système	Initialise la phase d'implémentation du système et le modèle d'implémentation du système correspondant à partir du modèle de conception du système et des modèles d'analyse des plates-formes. **Réalisation :** chef de projet (automatisable) **Pré-condition :** modèle de conception du système CONSISTANT ou COMPLET **Post-condition :** modèle d'implémentation du système INITIALISÉ



Explication : initialement, les modèles de *conception du système* et les modèles d'*analyse des plates-formes* hébergeant la partie applicative du système sont CONSISTANTS ou COMPLETS. Cette tâche *initialise* le modèle d'*implémentation du système*. Il contient le *système* du modèle de *conception du système* composant les plates-formes. Ces dernières contiennent les mondes, conteneurs, ressources et périphériques qui les composent. Le modèle reprend également les *acteurs* du modèle de *conception du système*. Dans l'état initial, les fragments d'objet du modèle de *conception du système* ne sont pas encore importés dans le modèle d'*implémentation du système*.

TABLE 85 – Gestion de projet : Tâche d'initialisation de la phase d'implémentation du système

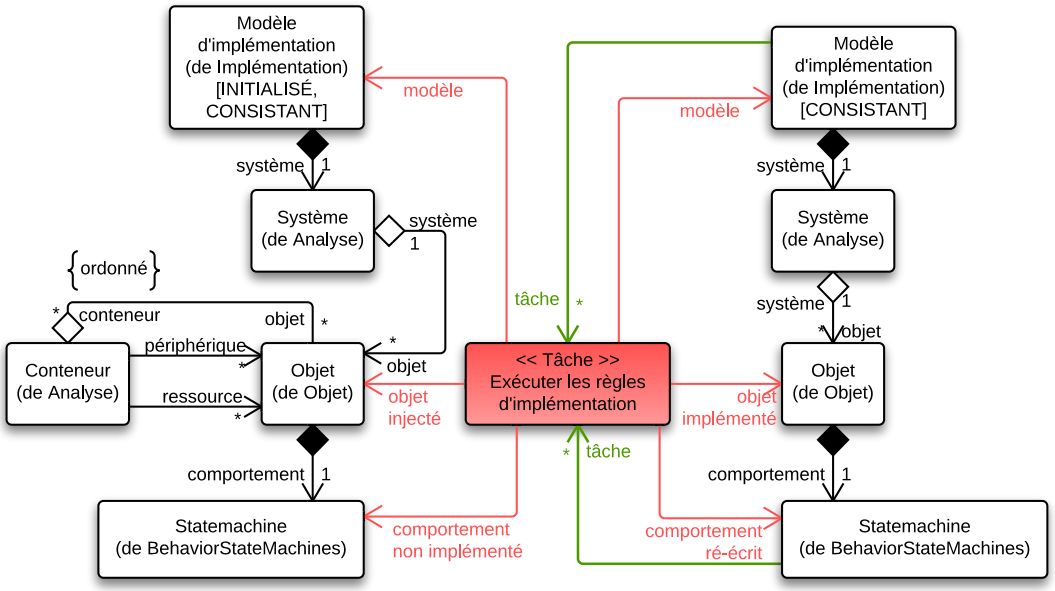
Tâche	Description
Injecter un objet dans un conteneur	Injecte un objet de conception (i.e. un fragment d'objet d'analyse hébergé dans un monde) dans les conteneurs de ce monde. **Réalisation :** développeur **Pré-condition :** modèle d'implémentation du système INITIALISÉ ou CONSISTANT **Post-condition :** au moins un fragment d'objet a été implémenté dans un conteneur



Explication : la tâche d'injection d'un objet dans un conteneur consiste à choisir un fragment d'objet hébergé dans un monde d'une plate-forme et l'injecter dans un ou plusieurs conteneurs offerts par ce monde. Pour réaliser cette tâche, le modèle d'*implémentation du système* peut être dans l'état INITIALISÉ ou CONSISTANT. À l'issue de cette tâche, le modèle contient le fragment d'objet hébergé dans le monde et *injecté* dans un ou plusieurs conteneurs, mais non encore implémenté. Cette tâche constituant la première partie de l'activité d'*implémentation* (l'autre partie étant l'*exécution des règles d'implémentation* des conteneurs, selon la figure 73), elle n'altère pas l'état du modèle d'implémentation du système.

TABLE 86 – Gestion de projet : Tâche d'injection d'un objet dans les conteneurs

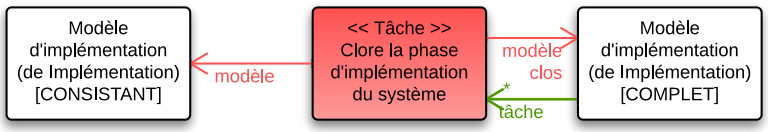
Tâche	Description
Exécuter les règles d'implémentation	Exécute les règles d'implémentation du conteneur sur l'objet injecté durant la tâche de *injection d'un objet dans un conteneur* **Réalisation :** développeur **Pré-condition :** modèle d'implémentation du système INITIALISÉ ou CONSISTANT **Post-condition :** modèle d'implémentation du système CONSISTANT



Explication : l'injection d'un objet dans un ou plusieurs conteneurs a été obtenue durant la tâche d'*injection d'un objet dans un conteneur* qui précède la tâche d'*exécution des règles d'implémentation*. Cette dernière consiste à exécuter les règles de chaque conteneur dans lequel l'objet a été injecté afin de ré-écrire son comportement et ses associations. L'objet *injecté* devient un objet *implémenté* accédant aux ressources et périphériques de la plate-forme. Le modèle d'*implémentation du système* devient CONSISTANT et le ou les conteneurs ayant reçu l'objet sont *remplacés* par l'objet *implémenté*.

TABLE 87 – Gestion de projet : Tâche d'exécution des règles d'implémentation

Tâche	Description
Clore la phase d'implémentation du système	Clôt la phase d'implémentation du système. **Réalisation :** chef de projet (automatisable) **Pré-condition :** modèle de conception du système COMPLET, modèle d'implémentation du système CONSISTANT **Post-condition :** modèle d'implémentation du système COMPLET



Explication : cette tâche permet de clore la phase d'*implémentation du système*. Cette phase peut être close lorsque la phase de *conception du système* pour le développement de la partie applicative du système et la phase d'*analyse du système* pour le développement de chaque plate-forme du système sont elles-mêmes closes et que toutes les tâches d'implémentation du système ont permis de satisfaire tous les scénarios du modèle d'*analyse du besoin* en phase d'*implémentation du système*. La tâche de clôture de la phase d'*implémentation du système* permet de *compléter* le modèle d'implémentation du système qui devient COMPLET.

TABLE 88 – Gestion de projet : Tâche de clôture de la phase d'implémentation du système

Annexe D

Classe *Object* et son constructeur en Arduino

```
#ifndef Object_H
  #define Object_H

#include <mthread.h>

class Object : public Thread {
  public:
    Object (uint8_t nb_block, uint8_t block_size);
    void write(void * value);
    bool is_empty();
    void init();
  protected:
    void* read();
    virtual bool loop() = 0;
    bool is_overrun();
  private:
    int buffer_size;
    uint8_t nb_block;
    uint8_t block_size;

    uint8_t * buffer;
    uint8_t * buffer_bound_high;
    uint8_t * buffer_bound_low;
    uint8_t * buffer_pointer_read;
    uint8_t * buffer_pointer_write;
    bool flag_overrun;
    bool flag_empty;
};
#endif

Object::Object (uint8_t nb_block, uint8_t block_size) {
    this->nb_block = nb_block;
    this->block_size = block_size;
    this->buffer_size = nb_block * block_size;
    this->buffer = (uint8_t *) malloc(buffer_size);
    for (uint8_t i = 0; i < nb_block; i++) {
      buffer[i] = 0;
    }

    this->buffer_pointer_read = (uint8_t *) buffer;
    this->buffer_pointer_write = (uint8_t *) buffer;
    this->buffer_bound_low = (uint8_t *) buffer;
    this->buffer_bound_high =  buffer_bound_low + this->buffer_size - 1;
    this->flag_overrun = false;
    this->flag_empty = true;
}
```

Listing D.1 – Déclaration de la classe *Object* et implémentation de son constructeur

Le listing D.1 présente la déclaration de la classe *Object* et l'implémentation de son constructeur. Le constructeur prend en paramètre le nombre de blocs et la taille d'un bloc (i.e. la taille d'un message). À partir de ces deux informations, il alloue un espace de la mémoire avec la commande `this->buffer = (uint8_t *) malloc(buffer_size);`. L'objet fournit deux drapeaux permettant de vérifier si la file est vide (*empty*) ou si des données sont perdues dans la file (*overrun*).

Cette classe non générée a été utilisée dans le cadre de la génération de code sur la plateforme Arduino UNO dans la section 3 du chapitre 10. Il est générique et permet d'adapter la taille de la file de réception à chaque objet.

Annexe E

Template Acceleo pour la génération de la fonction loop

Ce template permet de générer la méthode *loop ()* des objets de la plate-forme Arduino UNO. Elles se décomposent en plusieurs étapes :

1. Préalablement, la méthode vérifie que la file n'est pas vide :
   this−>isEmpty ()

2. On lit un premier message :
   struct message ∗ m = (struct message ∗) this−>read();

3. Pour toutes les régions et tous les états courants des régions, on vérifie le message
   case ___HOE2_[trigger.event.oclAsType(SignalEvent).signal.name.normalizeUp()/]___:

4. On teste la garde :
   [trigger .owner.oclAsType(Transition).generateConstraint(object)/]

5. Si nécessaire, on s'enregistre ou se dés-enregistre en tant qu'écouteur de messages si nécessaire :
   [state . unregistrerAsListener () /]
   [trigger .owner.oclAsType(Transition).target.oclAsType(State).registrerAsListener()/]

6. On interprète le signal reçu :
   [trigger .event.oclAsType(SignalEvent).interpretSignalEvent()/]

7. On effectue l'activité lors du franchissement de la transition :
   [trigger .owner.oclAsType(Transition).doActivity()/]

8. On franchit la transition et on change d'état :
   this−>currentState[region.name.normalize()/] = ___HOE2_[trigger.owner.oclAsType(Transition).target .name.normalizeUp()/]___;

```
[comment Generate implementation for one object /]
[template public generateImplementationFile(object : Object) post (trim())]
[comment ...]
bool [object.name.normalizeUpFirst()/]::loop () {
  if (this->isEmpty ()) {
  }
  else {
    do {
      struct message * m = (struct message *) this->read();
      [for (region : Region | object.behavior.region)]
      // for region [object.behavior.region.name.normalizeUpFirst()/]
      switch (this->currentState[region.name.normalize()/]) {
      [for (state : State | region.subvertex->selectByType(State))]
        case __HOE2_[state.name.normalize()/]__:
          switch (m->messageID) {
          [for (trigger : Trigger | state.outgoing.trigger)]
            [if (trigger.event.oclIsKindOf(SignalEvent))]
            case __HOE2_[trigger.event.oclAsType(SignalEvent).signal.name.
    normalizeUp()/]__:
              [trigger.owner.oclAsType(Transition).generateConstraint(object)/]
              [state.unregistrerAsListener()/]
              [trigger.event.oclAsType(SignalEvent).interpretSignalEvent()/]
              [trigger.owner.oclAsType(Transition).doActivity()/]
              this->currentState[region.name.normalize()/] = __HOE2_[trigger.
    owner.oclAsType(Transition).target.name.normalizeUp()/]__;
              [trigger.owner.oclAsType(Transition).target.oclAsType(State).
    registrerAsListener()/]
            break;
            [/if]
          [/for]
            default:
            break;
          }
        break;
      [/for]
        default:
        break;
      }
      [/for]
    } while(!this->isEmpty());
  }
  sleep_milli (1);
}
[comment ...]
[/template]
```

Listing E.1 – Génération de la fonction loop

Annexe F

Template Acceleo pour la génération du patron observateur

Afin de concevoir l'interpréteur de machine à états dans le chapitre 10 permettant d'interpréter les machines à états du lanagge ⟨HOE⟩[2]. Il est nécessaire de prendre en compte toutes les structures de construction des machines à états. Dans cette annexe, nous présentons les templates Acceleo permettant la prise en compte du patron *Observateur* et le résultat généré.

Implémentation du patron « Observateur ».

```
[comment Get the property targeted by an association /]
[query public getTargetFromAssociation(asso : Association) : Property = asso.
    memberEnd->any(p | asso.ownedEnd->includes(p)._not()) /]

[template public registrerAsListener(state : State)]
  [for (t : Trigger | state.outgoing.trigger)]
    [if (t.event.oclIsTypeOf(ListenEvent))]
    [let se : ListenEvent = t.event.oclAsType(ListenEvent)]
    message_prepared.initiator = (int)this;
    message_prepared.messageID = __HOE2_[se.signal.name.normalizeUp()/]__;
    [se.notifier.type.oclAsType(Association).getTargetFromAssociation().name.
    normalizeDownFirst()/]->set_[se.signal.name.normalizeDownFirst()/]_listener(
    this, (void *)&message_prepared);
    [/let]
    [/if]
  [/for]
[/template]

[template public unregistrerAsListener(state : State)]
  [for (t : Trigger | state.outgoing.trigger)]
    [if (t.event.oclIsTypeOf(ListenEvent))]
    [let se : ListenEvent = t.event.oclAsType(ListenEvent)]
    [se.notifier.type.oclAsType(Association).getTargetFromAssociation().name.
    normalizeDownFirst()/]->unset_[se.signal.name.normalizeDownFirst()/]
    _listener();
    [/let]
    [/if]
  [/for]
[/template]
```

Listing F.1 – Implémentation du patron « Observateur »

Les deux templates *registrerAsListener ()* et *unregistrerAsListener ()* présente notre implémentation spécifique du patron « *Observateur* » pour la plate-forme Arduino UNO. Le listing F.1 illustre ces deux templates. Le template *registrerAsListener ()* permet à un objet

lorsqu'il franchit une transition de s'enregistrer en tant qu'écouteur d'un message sur une transition sortante du nouvel état courant. Lorsque qu'il quitte ce nouvel état, il appelle la méthode *unregistrerAsListener ()* afin de ne plus écouter.

```
case __HOE2_ABSENCE__:
  switch (m->messageID) {
    case __HOE2_HIGH_SET__:
      // Désinscription de l'écoute du message high_set de la broche
      pin2->unset_high_set_listener();
      // Préparation du message que la broche doit lui envoyer dans le cas
      // d'une notification du message low_set
      message_prepared.initiator = (int)this;
      message_prepared.messageID = __HOE2_LOW_SET__;
      // Inscription de l'écoute du message low_set de la broche
      pin2->set_low_set_listener(this, (void *)&message_prepared);
      // Changement d'état
      this->currentStateregion1 = __HOE2_PRESENCE__;
    break;
```

Listing F.2 – Génération du code de l'*écouteur* du patron « Observateur »

Le résultat produit par ces deux templates sont illustrées par le listing F.2. Il présente une partie de l'interpréteur de la machine à états du capteur de présence PIR. Initialement, cet objet *écoute* le (il s'est enregistré en tant qu'écouteur du) message *high_set* de la broche. Lorsque celle-ci le notifie, il se désinscrit de cette écoute et s'inscrit désormais en tant qu'écouteur du message *low_set*. L'inscription est très particulière et spécifique à la plate-forme. L'objet jouant le rôle de *notifieur* connait l'objet (puisqu'il s'est enregistrer) à qui renvoyer le message, mais ne connait le format du message attendu par l'objet *notifieur* (pour rappel, la taille du message est différente pour chaque objet). Ainsi, lorsqu'un objet s'inscrit en tant qu'écouteur, il doit lui transmettre un message que l'objet *notifieur* lui renverra alors au moment venu. Ainsi, l'objet *notifieur* ne s'occupe ni de l'objet qui écoute le message, ni le message qu'il est en charge de lui renvoyer. Le listing F.3 illustre la prise en charge, et le renvoi du message à l'objet *écouteur*.

```
#ifndef Pin_H
  #define Pin_H

#include <Object.h>
class Pin : public Object {
  public:
    void set_low_set_listener (Object * object, void * m_prepared);
    void unset_low_set_listener ();
    void set_high_set_listener (Object * object, void * m_prepared);
    void unset_high_set_listener ();
  private:
    Object * low_set_listener;
    void * m_prepared_low_set;
    Object * high_set_listener;
    void * m_prepared_high_set;
};

#endif

void Pin::set_low_set_listener (Object * object, void * m_prepared) {
  this->low_set_listener = object;
  this->m_prepared_low_set = m_prepared;
```

```
}
void Pin::unset_low_set_listener () {
  this->low_set_listener = NULL;
  this->m_prepared_low_set = NULL;
}
void Pin::set_high_set_listener (Object * object, void * m_prepared) {
  this->high_set_listener = object;
  this->m_prepared_high_set = m_prepared;
}
void Pin::unset_high_set_listener () {
  this->high_set_listener = NULL;
  this->m_prepared_high_set = NULL;
}
```

Listing F.3 – Génération du code du *notifieur* du patron « Observateur »

Pour la plate-forme Arduino UNO, nous avons fait l'hypothèse qu'un seul écouteur pouvait s'enregistrer à la fois. Cette hypothèse qui n'est pas valable pour toutes les implémentations, est suffisante pour la plate-forme Arduino UNO, dans la mesure où le seul cas d'utilisation du patron *observateur* se situe à l'écoute des broches et que nous avons considéré qu'une broche n'était accessible que par un seul objet, selon le principe de responsabilité unique évoqué dans le chapitre 6.

```
case __HOE2_LOW__:
  switch (m->messageID) {
    case __HOE2_VCC__:
      // Génération de la contrainte oclIsInState(INPUT)
      if(this->currentStateregion1 != __HOE2_INPUT__ && this->
    currentStateregion2 != __HOE2_INPUT__)
        break;
      // Si un écouteur est bien enregistré
      if (this->high_set_listener != NULL) {
        // Retournement de l'objet préparé par l'objet écouteur
        this->high_set_listener->write(this->m_prepared_high_set);
      }
      // Changement d'état
      this->currentStateregion2 = __HOE2_HIGH__;
      break;
```

Listing F.4 – Envoi du message à l'*écouteur*

Le listing F.4 illustre un fragment de l'interprétation de la machine à états de la broche. Dans le cas où la broche est dans les états concurrents LOW et INPUT et qu'il reçoit le message vcc [1], alors il retourne à l'objet *écouteur* le message *m_prepared_high_set* que l'objet *écouteur* lui avait fournit au moyen de la méthode `void set_high_set_registrer (Object * listener, void * m_prepared)`.

1. Ce message est un message propre à la plate-forme.

Glossaire

AADL Architecture Analysis and Design Language. 3

Acceleo . 185, 221, 223–225, 229–231, 234, 275

Arduino UNO . xi, xiii, 130, 131, 133, 135, 136, 139–142, 148, 151, 211, 212, 214–216, 219, 221, 223–226, 228, 229, 231–234, 272, 273, 275, 277

ASIC Application-Specific Integrted Circuit (ASIC) désigne un circuit intégré dédié à une application particulière. 133

ATL Atlas Transformation Language. 185, 214

BIP Behavior, Interaction, Priority. x, 39, 47, 54–57

BIRT Business Intelligent and Reporting Tools. 198

BPMN Business Process Model and Notation. 36

CBD Component-based design. 51

CEA Commissariat à l'énergie atomique et aux énergies alternatives. vi, 182, 185

COTS Commercial Off-The-Shelf. 29, 32

CSI Camera Serial Interface. 212

CSS Cascading Style Sheet. 185

DSL Domain Specific Language. 72

Eclipse . 61, 72, 82, 177, 180, 182–185, 192, 193, 197, 207, 279, 281

Eclipse MDT . 182, 184, 185, 189, 214

Ecore Implémentation du méta-méta-modèle MOF pour l'environnement de développement intégré Eclipse. 182, 184–186, 281

EMF Eclipse Modeling Framework. 182, 184, 185, 189, 198, 214

EPL Eclipse Public License. 180

Equinox Implémentation de la spécification OSGi pour l'environnement de développement intégré Eclipse. 180

ETL Eclipse Transformation Language. 185, 214

EWL Eclipse Wizard Language. 214, 215, 220, 234

Gantt Un diagramme de Gantt est un type de diagrammes à bar développé par Henry Gantt dans les années 1910 pour illustrer . 81, 197, 198, 202

Git Logiciel de gestion de versions décentralisés développé par Linux Tovarlds en 2005. 83, 183, 203–207

GMF Graphical Modeling Framework. xii, 184–186, 188, 189, 191

Graphiti . 184–189, 191–193, 200, 207

HDMI High-Definition Multimedia Interface. 211

I²C Inter Integrated Circuit. 108, 109, 122, 123, 125, 127, 139–142, 145, 147, 150, 212, 218

IP-XACT . 3

IUT Institut universitaire de technologie. v

Java . 186, 189, 200, 207, 214, 229

LDAP Lightweight Directory Access Protocol. 183, 194, 196, 198, 199

LIALP Laboratoire infrastructure et atelier logiciel pour puces. v, 182

LIG Laboratoire d'informatique de Grenoble. v, vi

MARTE Modeling and Analysis of Real-Time Embedded Systems. 57, 58, 60, 61, 237, 238

MDA Model-Driven Architecture. 57

MDT Model Development Tools. 182, 184, 185, 189, 214, 279

Mercurial Logiciel de gestion de versions décentralisé, développé par Matt Mackall en 2005. 203, 204

MOF Meta-Object Facility. 35, 182, 279

NoC Network-on-Chip. 97, 124

OCL Object Constraint Language. 87, 187, 214, 229, 251, 252, 254

OMG Object Management Group. 58, 281

OpenCV . 135

OSGi Framework implémentant un modèle à composants dont le cycle de vie de chaque composant (déploiement, installation, démarrage, mis à jour, arrêt, désinstallation est dynamiquement gérée et ne nécessite pas l'arrêt et le redémarrage de la plate-forme Java.. 180, 279

Papyrus MDT . xii, 180, 181, 184, 185, 187–189, 191, 198, 207

PDSI Processus de développement de systèmes industriels. 71, 104, 249

PIM Platform Independent Model. 57

PIR Peripheral InfraRed. xiii, 136, 140–142, 147, 151, 212, 215, 218, 223, 224, 276

plug-in En français « *greffon* », module interagissant avec un programme principal. 180

PMBOK Project Management Body of Knowledge. xii, 154–156, 169

PMI Project Management Institute. 154

PSM Platform Specific Model. 57

Python . 135, 212

RAD Rapid Development Application. 30, 159

Raspberry PI . xi, xiii, 133, 135, 144–148, 150, 211–214, 216–219, 221

RSA Rational Software Architect. 35

RUP Rational Unified Process. 31, 71, 155, 156

SD CARD . 211

SIGMA Systèmes d'Information - inGénierie et Modélisation Adaptables. v, vi, 182

SoC System on Chip. 47

SOI System of Interest. 73

SPEM Software & Systems Process Engineering Metamodel. 32, 35–37

SPI Serial Peripheral Interface. 212

Spray . 186, 187

SPT Schedulabilty Performance and Time. 237

SRAM Static Random Access Memory. 221, 228, 231

SRP Single Responsibility Principle. 122

Subversion Logiciel de gestion de versions centralisé développé par la fondation Apache en 2000. 203, 204

SWEBOK Software Engineering Body of Knowledge. 155

SWT Standard Widget Toolkit. 197, 198

SysML System Modeling Language. 57, 60, 61

TTM Time-to-Market. 28

UML Unified Modeling Language. 3, 31, 34–37, 48, 57, 58, 60, 72, 84, 86, 89–92, 94–97, 100, 102, 103, 106, 157, 158, 160–164, 168, 170, 178, 181, 184–189, 191, 197, 198, 225, 238, 246, 253, 254, 281

UML2 Implémentation basé sur le méta-modèle Ecore des méta-modèles UML 2.x définis par l'OMG pour la plate-forme Eclipse. xii, 184–191, 198, 207

USB Universal Serial Bus. 108, 139–142, 145, 147, 211, 212

USDP Unified Software Development Process. 31, 71, 159

VHDL VHSIC Hardware Description Language (VHDL) est un langage de description de matériel. 133, 238

Xtend . 185

Bibliographie

[Abran *et al.* 2001] Alain Abran, Pierre Bourque, Robert Dupuis et James W Moore. Guide to the software engineering body of knowledge-swebok. IEEE Press, 2001.

[Acuña & Ferré 2001] Silvia Teresita Acuña et Xavier Ferré. *Software Process Modelling.* In ISAS-SCI (1), pages 237–242, 2001.

[Akehurst *et al.* 2007] D. H. Akehurst, O. Uzenkov, W. G. Howells, K. D. Mcdonald-Maier et B. Bordbar. *An Experiment in Using Model Driven Development : Compiling UML State Diagrams into VHDL.* In Forum on Specification and Design Languages (FDL'07), Septembre 2007.

[Arduino Official Website 2014] Arduino Official Website, 2014.

[Assayad 2009] Ismail Assayad. *A Platform-based Design Framework for Joint SW/HW Multiprocessor Systems Design.* J. Syst. Archit., vol. 55, no. 7-9, pages 409–420, Juillet 2009.

[Augsburg 2005] Tanya. Augsburg. Becoming Interdisciplinary : an Introduction to Interdisciplinary Studies. Kendall/Hunt Pub., Dubuque, Iowa, 2005.

[Aulagnier *et al.* 2009] Denis Aulagnier, Ali Koudri, Stéphane Lecomte, Philippe Soulard, Joël Champeau, Jorgiano Vidal, Gilles Perrouin et Pierre Leray. *SoC/SoPC development using MDD and MARTE profile.* In Jean-Philippe Babau, Mireille Blay-Fornarino, Joël Champeau, Sèbastien Gèrard, Sylvain Robert et Antonino Sabetta, editeurs, Model Driven Engineering for Distributed Real-time Embedded Systems. ISTE, Mars 2009.

[Balarin *et al.* 2002] Felice Balarin, Luciano Lavagno, Claudio Passerone, Alberto Sangiovanni-Vincentelli, Marco Sgroi et Yosinori Watanabe. *Modeling and Designing Heterogeneous Systems.* In Jordi Cortadella, Alex Yakovlev et Grzegorz Rozenberg, editeurs, Concurrency and Hardware Design, volume 2549 of *Lecture Notes in Computer Science*, pages 228–273. Springer Berlin Heidelberg, 2002.

[Balarin *et al.* 2003] F. Balarin, Y. Watanabe, H. Hsieh, L. Lavagno, C. Passerone et A. Sangiovanni-Vincentelli. *Metropolis : an integrated electronic system design environment.* Computer, vol. 36, no. 4, pages 45–52, Avril 2003.

[Basu *et al.* 2011] A. Basu, S. Bensalem, M. Bozga, J. Combaz, M. Jaber, Thanh-Hung Nguyen et J. Sifakis. *Rigorous Component-Based System Design Using the BIP Framework.* Software, IEEE, vol. 28, no. 3, pages 41–48, Avril 2011.

[Beck *et al.* 2001] Kent Beck, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Steve Mellor, Ken Schwaber, Jeff Sutherland et Dave Thomas. *Manifesto for Agile Software Development*, 2001. `http://agilemanifesto.org`.

[Bendraou *et al.* 2007] R. Bendraou, B. Combemale, X. Cregut et M.-P. Gervais. *Definition of an Executable SPEM 2.0.* In Software Engineering Conference, 2007. APSEC 2007. 14th Asia-Pacific, pages 390–397, Dec 2007.

[Benediktsson *et al.* 2003] Oddur Benediktsson, Darren Dalcher, Karl Reed et Mark Woodman. *COCOMO-Based Effort Estimation for Iterative and Incremental Software Development*. Software Quality Journal, vol. 11, no. 4, pages 265–281, Novembre 2003.

[Bensalem *et al.* 2011] Saddek Bensalem, Andreas Griesmayer, Axel Legay, Thanh-Hung Nguyen, Joseph Sifakis et Rongjie Yan. *D-Finder 2 : Towards Efficient Correctness of Incremental Design*. In Mihaela Bobaru, Klaus Havelund, GerardJ. Holzmann et Rajeev Joshi, editeurs, NASA Formal Methods, volume 6617 of *Lecture Notes in Computer Science*, pages 453–458. Springer Berlin Heidelberg, Avril 2011.

[Biehl *et al.* 2012] Matthias Biehl, Jiarui Hong et Frederic Loiret. *Automated Construction of Data Integration Solutions for Tool Chains*. In The Seventh International Conference on Software Engineering Advances, Novembre 2012.

[Biehl 2010] M. Biehl. *Supporting Model Evolution in Model-Driven Development of Automotive Embedded Systems*. PhD thesis, KTH Royal Institute of Technology, Stockholm, Sweden, 2010, Novembre 2010.

[Biehl 2011] Matthias Biehl. *Tool Integration Language (TIL)*. Rapport technique 2011 :14, KTH, Mechatronics, Septembre 2011. QC 20111130.

[Boehm 1988] B. W. Boehm. *A Spiral Model of Software Development and Enhancement*. Computer, vol. 21, no. 5, pages 61–72, Mai 1988.

[Booch 1994] Grady Booch. Object-Oriented Analysis and Design with Applications. Addison Wesley Longman, 1994.

[Bourque *et al.* 2002] Pierre Bourque, François Robert, Jean-Marc Lavoie, Ansik Lee, Sylvie Trudel et Timothy C Lethbridge. *Guide to the software engineering body of knowledge (swebok) and the software engineering education knowledge (seek)-a preliminary mapping*. In Software Technology and Engineering Practice, 2002. STEP 2002. Proceedings. 10th International Workshop on, pages 8–23. IEEE, 2002.

[Broy 2003] M. Broy. *Modular hierarchies of models for embedded systems*. In First ACM and IEEE International Conference on Formal Methods and Models for Co-Design, 2003. MEMOCODE '03, pages 183–195, Juin 2003.

[Broy 2006a] M. Broy. *The 'Grand Challenge' in Informatics : Engineering Software-Intensive Systems*. Computer, vol. 39, no. 10, pages 72–80, Octobre 2006.

[Broy 2006b] Manfred Broy. *Challenges in automotive software engineering*. In Proceedings of the 28th international conference on Software engineering, ICSE '06, pages 33–42, New York, NY, USA, Mai 2006. ACM.

[Broy 2009] Manfred Broy. *Seamless Model Driven Systems Engineering Based on Formal Models*. In Karin Breitman et Ana Cavalcanti, editeurs, Formal Methods and Software Engineering, volume 5885 of *Lecture Notes in Computer Science*, pages 1–19. Springer Berlin Heidelberg, Décembre 2009.

[Callegari & Bastos 2007] D.A Callegari et R.M. Bastos. *Project Management and Software Development Processes : Integrating RUP and PMBOK*. In Systems Engineering and Modeling, 2007. ICSEM '07. International Conference on, pages 1–8, March 2007.

[Céret *et al.* 2013] Eric Céret, Sophie Dupuy-Chessa, Gaëlle Calvary, Agnès Front et Dominique Rieu. *A taxonomy of design methods process models*. Information and Software Technology, vol. 55, no. 5, pages 795–821, Novembre 2013.

[Cottrell 2004] Bill Cottrell. *Standards, compliance, and Rational Unified Process, Part I : Integrating RUP and the PMBOK*. IBM Developerworks,(May 10, 2004), 2004.

[Curtis *et al.* 1992] Bill Curtis, Marc I. Kellner et Jim Over. *Process Modeling*. Commun. ACM, vol. 35, no. 9, pages 75–90, Septembre 1992.

[Duncan 1996] William R Duncan. *A guide to the project management body of knowledge*. 1996.

[Dupuy-Chessa 2011] S. Dupuy-Chessa. Modélisation en Interaction Homme-Machine et en Système d'Information : A la croisée des chemins. Master's thesis, Université de Grenoble, Décembre 2011.

[Ellner *et al.* 2010] Ralf Ellner, Samir Al-Hilank, Johannes Drexler, Martin Jung, Detlef Kips et Michael Philippsen. *eSPEM – a SPEM Extension for Enactable Behavior Modeling*. In Proceedings of the 6th European Conference on Modelling Foundations and Applications, ECMFA'10, pages 116–131, Berlin, Heidelberg, 2010. Springer-Verlag.

[Ernst *et al.* 1993] R. Ernst, J. Henkel et T. Benner. *Hardware-software cosynthesis for microcontrollers*. Design Test of Computers, IEEE, vol. 10, no. 4, pages 64–75, Décembre 1993.

[Ferrari & Sangiovanni-Vincentelli 1999] A. Ferrari et A. Sangiovanni-Vincentelli. *System Design : Traditional Concepts and New Paradigms*. In International Conference on Computer Design, pages 2–12, Octobre 1999.

[Finkelstein *et al.* 1994] Anthony Finkelstein, Jeff Kramer et Bashar Nuseibeh. Software process modelling and technology. John Wiley & Sons, Inc., 1994.

[Fraser *et al.* 1994] Martin D. Fraser, Kuldeep Kumar et Vijay K. Vaishnavi. *Strategies for Incorporating Formal Specifications in Software Development*. Commun. ACM, vol. 37, no. 10, pages 74–86, Octobre 1994.

[Fuggetta 2000] Alfonso Fuggetta. *Software Process : A Roadmap*. In Proceedings of the Conference on The Future of Software Engineering, ICSE '00, pages 25–34, New York, NY, USA, 2000. ACM.

[Gamatié *et al.* 2011] Abdoulaye Gamatié, Sébastien Le Beux, Éric Piel, Rabie Ben Atitallah, Anne Etien, Philippe Marquet et Jean-Luc Dekeyser. *A Model-Driven Design Framework for Massively Parallel Embedded Systems*. ACM Trans. Embed. Comput. Syst., vol. 10, no. 4, pages 39 :1–39 :36, Novembre 2011.

[Gamma *et al.* 1994] Erich Gamma, Richard Helm, Ralph Johnson et John Vlissides. Design patterns : elements of reusable object-oriented software. Pearson Education, 1994.

[Gartner 2013] Gartner. *Gartner Says Smartphone Sales Grew 46.5 Percent in Second Quarter of 2013 and Exceeded Feature Phone Sales for First Time*, Août 2013. http ://www.gartner.com/newsroom/id/2573415.

[Gérard 2000] Sébastien Gérard. *Modélisation UML exécutable pour les systèmes embarqués de l'automobile*. PhD thesis, Université d'Evry, Octobre 2000.

[Gilb 1976] T. Gilb. Software metrics. Winthrop computer systems series. Winthrop Publishers, 1976.

[Gilb 1985] Tom Gilb. *Evolutionary Delivery Versus the "Waterfall Model"*. SIGSOFT Softw. Eng. Notes, vol. 10, no. 3, pages 49–61, Juillet 1985.

[Godart & Perrin 2009] Claude Godart et Olivier Perrin. Les processus métiers : Concepts, modèles et systèmes. Traité Informatique et Systèmes d'Information, IC2. Hermes, Mai 2009.

[Graf 1999] R.F. Graf. Modern Dictionary of Electronics. Electronics & Electrical. Newnes, 1999.

[Graham 1989] D.R Graham. *Incremental Development : Review of nonmonolithic life-cycle development methods.* Information and Software Technology, vol. 31, no. 1, Janvier 1989.

[Guide 2001] A Guide. *PROJECT MANAGEMENT BODY OF KNOWLEDGE (PMBOK® GUIDE).* In Project Management Institute, 2001.

[Gupta & De Micheli 1993] R. K. Gupta et G. De Micheli. *Hardware-Software Cosynthesis for Digital Systems.* Design & Test of Computers, IEEE, vol. 10, no. 3, pages 29–41, Septembre 1993.

[Harmsen 1997] Anton Frank Harmsen. *Situational Method Engineering.* PhD thesis, University of Twente, Janvier 1997.

[Heineman & Councill 2001] G.T. Heineman et W.T. Councill. *Component-Based Software Engineering.* In Addison-Wesley, 2001.

[Henzinger & Sifakis 2007] Thomas A. Henzinger et Joseph Sifakis. *The Discipline of Embedded Systems Design.* Computer, vol. 40, no. 10, pages 32–40, Octobre 2007.

[Highsmith & Fowler 2001] Jim Highsmith et Martin Fowler. *The Agile Manifesto.* Software Development Magazine, vol. 9, no. 8, pages 29–30, 2001.

[Hili *et al.* 2012a] Nicolas Hili, Christian Fabre, Sophie Dupuy-Chessa et Stéphane Malfoy. *Efficient Embedded System Development : A Workbench for an Integrated Methodology.* In ERTS2 2012, Toulouse, France, Février 2012.

[Hili *et al.* 2012b] Nicolas Hili, Christian Fabre, Dominique Rieu, Sophie Dupuy-Chessa et Stéphane Malfoy. $\langle HOE \rangle^2$*, une méthode intégrée pour le développement des systèmes embarqués.* In XXXème Congrès Inforsid'2012, Mai 2012.

[Hili *et al.* 2014a] Nicolas Hili, Christian Fabre, Sophie Dupuy-Chessa et Dominique Rieu. *A Model-Driven Approach for Embedded System Prototyping and Design.* In IEEE International Symposium on Rapid System Prototyping (RSP'14) (part of ESWEEK'14), New Dehli, India, Octobre 2014.

[Hili *et al.* 2014b] Nicolas Hili, Christian Fabre, Sophie Dupuy-Chessa, Dominique Rieu et Ivan Llopard. *Model-Based Platform Composition for Embedded System Design.* In IEEE 8th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC-14) (IEEE MCSoC-14), Aizu-Wakamatsu, Japan, Septembre 2014.

[Hili 2013a] N. Hili. *How to Efficiently and Rapidly Build Graphical Modelling Editors.* Xerox Research Centre Europe, Novembre 2013. Eclipse DemoCamps. `http://wiki.eclipse.org/Eclipse_DemoCamps_November_2013/Grenoble`.

[Hili 2013b] Nicolas Hili. *Diviser pour mieux régner : le cas du développement des systèmes embarqués.* In 16ème Journées Nationales du Réseau Doctoral en Micro-Nanoélectronique, Grenoble,France, Juin 2013.

[IBM 2003] IBM. *Rational Unified Process : Best Practices for Software Development Teams*, Décembre 2003.

[INCOSE 2011] INCOSE. *INCOSE Systems Engineering Handbook v3.2.2*, Octobre 2011.

[INCOSE 2014] INCOSE, Janvier 2014.

[ISO 2009] ISO. *ISO/DIS 26262-1 - Road vehicles — Functional safety — Part 1 Glossary.* Rapport technique, Geneva, Switzerland, Juillet 2009.

[Ivar Jacobson *et al.* 1999] Ivar Jacobson, Grady Booch et James Rumbaugh. The Unified Software Development Process. Addison Wesley, Février 1999.

[Keutzer *et al.* 2000] K. Keutzer, A.R. Newton, J.M. Rabaey et A. Sangiovanni-Vincentelli. *System-Level Design : Orthogonalization of Concerns and Platform-Based Design.* IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 19, no. 12, pages 1523–1543, Décembre 2000.

[Kienhuis *et al.* 2002] Bart Kienhuis, Ed Deprettere, Pieter van der Wolf et Kees Vissers. *A Methodology to Design Programmable Embedded Systems.* In Ed Deprettere, Jürgen Teich et Stamatis Vassiliadis, editeurs, Embedded Processor Design Challenges, volume 2268 of *Lecture Notes in Computer Science*, pages 321–324. Springer Berlin / Heidelberg, 2002. 10.1007/3-540-45874-3_2.

[Kirshin *et al.* 2007] Andrei Kirshin, Dolev Dotan et Alan Hartman. *A UML Simulator Based on a Generic Model Execution Engine.* In Thomas Kühne, editeur, Models in Software Engineering, volume 4364 of *Lecture Notes in Computer Science*, pages 324–326. Springer Berlin Heidelberg, 2007.

[Kopetz 1997] H. Kopetz. Real-time Systems : Design Principles for Distributed Embedded Applications. Kluwer international series in engineering and computer science. Kluwer Academic Publishers, 1997.

[Koudri *et al.* 2008] Ali Koudri, Denis Aulagnier, Didier Vojtisek, Philippe Soulard, Christophe Moy, Joël Champeau, Jorgiano Vidal et Jean-Christophe Le Lann. *Using MARTE in a Co-Design Methodology.* In Proceedings of MARTE Workshop at DATE, Munich, Allemagne, Mars 2008.

[Krakowiak 2009] Sacha Krakowiak. *Middleware Architecture with Patterns and Frameworks*, Février 2009.

[Lamothe 2013] J. Lamothe. *Arduino-Compatible Multi-Threading Library*, Octobre 2013.

[Larman & Basili 2003] Craig Larman et Victor R Basili. *Iterative and incremental development : A brief history.* Computer, vol. 36, no. 6, pages 47–56, 2003.

[Lee 2005] E.A. Lee. *Absolutely Positively On Time : What Would It Take ?* Computer, vol. 38, no. 7, pages 85–87, Juillet 2005.

[Llopard *et al.* 2014] Ivan Llopard, Albert Cohen, Christian Fabre et Nicolas Hili. *A Parallel Action Language for Embedded Applications and its Compilation Flow.* In 17th International Workshop on Software and Compilers for Embedded Systems, Sankt Goar, Allemagne, Juin 2014.

[Luiz Fernando Capretz 2005] Luiz Fernando Capretz. *Y : A New Component-Based Software Life Cycle Model.* Journal Of Computer Science, vol. 1, page 1, 2005.

[Martin 1991] James Martin. Rapid application development. Macmillan Publishing Co., Inc., Indianapolis, IN, USA, 1991.

[Martin 2002] Robert C Martin. *The single responsibility principle*. The Principles, Patterns, and Practices of Agile Software Development, pages 149–154, 2002.

[McDermid & Ripken 1983] John McDermid et Knut Ripken. *Life Cycle Support in the Ada Environment*. Ada Lett., vol. III, no. 1, pages 57–62, Juillet 1983.

[Nuseibeh 2001] B. Nuseibeh. *Weaving Together Requirements and Architectures*. Computer, vol. 34, no. 3, pages 115–119, Mars 2001.

[Obermaisser 2004] Roman Obermaisser. Event-triggered and time-triggered control paradigms, volume 22. Springer, 2004.

[Obj 2003] Object Management Group, Framingham, Massachusetts. *MDA Guide Version 1.0.1*, Juin 2003.

[Object Management Group 2005] Object Management Group. *UML Profile for Schedulability, Performance and Time v1.1*, Janvier 2005.

[Object Management Group 2010] Object Management Group. *Unified Modeling Language Superstructure 2.4.1*, Mai 2010.

[Object Management Group 2011] Object Management Group. *UML Profile for MARTE : Modeling and Analysis of Real-Time Embedded Systems v1.1*, Juin 2011.

[Object Management Group 2012] Object Management Group. *Systems Modeling Language Specification v1.3*, Juin 2012.

[Object Management Group 2013] Object Management Group. *Unified Modeling Language Superstructure 2.5 Beta*, Septembre 2013.

[OMG 2008] OMG. *Software & Systems Process Engineering Meta-Model Specification*, Avril 2008.

[OMG 2010] OMG. *Unified Modeling Language Specification 2.3*, Avril 2010.

[OMGL 1997] OMGL. *Analyse et conception orientées objets*. Équipe des systèmes d'information IUT2 Grenoble, Septembre 1997.

[Osterweil 1987] L. Osterweil. *Software Processes Are Software Too*. In Proceedings of the 9th International Conference on Software Engineering, ICSE '87, pages 2–13, Los Alamitos, CA, USA, 1987. IEEE Computer Society Press.

[Papyrus 2014] Papyrus. *Papyrus Home Page*, 2014. `http://eclipse.org/papyrus`.

[Pedroni 2013] V Pedroni. Index. MIT Press, 2013.

[Poulhiès *et al.* 2006] Marc Poulhiès, Jacques Pulou, Christophe Rippert et Joseph Sifakis. *A Methodology and Supporting Tools for the Development of Component-Based Embedded Systems*. In Composition of Embedded Systems. Scientific and Industrial Issues, volume 4888, page Lecture Notes in Computer Science, Paris, France, 2006.

[Project Management Institute 2014] Project Management Institute, 2014.

[Rafique-Golra 2014] Fahad Rafique-Golra. A refinement based methodology for software process modeling. Master's thesis, Téleécom Bretagne, Janvier 2014.

[Ramesh *et al.* 2012] U.B.K. Ramesh, S. Sentilles et I. Crnkovic. *Energy Management in Embedded Systems : Towards a Taxonomy*. In 2012 First International Workshop on Green and Sustainable Software, pages 41–44, Juin 2012.

[Raspberry PI Foundation 2014] Raspberry PI Foundation. *Raspberry PI Official Website*, 2014.

[Refsdal 2011] Ivar Refsdal. Comparison of gmf and graphiti based on experiences from the development of the prediqt tool. Master's thesis, University of Oslo, Novembre 2011.

[Rolland 1993] Colette Rolland. *Modeling the requirements engineering process.* In Information Modelling and Knowledge Bases V : Principles and Formal Techniques : Results of the 3rd European-Japanese Seminar, Budapest, Hungary, May, pages 85–96, 1993.

[Rolland 2005] Colette Rolland. *L'ingénierie des méthodes : une visite guidée.* e-TI, vol. 1, 2005.

[Royce 1970] Winston W Royce. *Managing the development of large software systems.* In proceedings of IEEE WESCON, volume 26. Los Angeles, 1970.

[Rygaert 2002] Bernard Rygaert. *PDSI : Processus de Développement des Systèmes Industriels*, 2002. Marketing flyer from SILICOMP.

[Sangiovanni-Vincentelli *et al.* 2000] Alberto Sangiovanni-Vincentelli, Marco Sgroi et Luciano Lavagno. *Formal Models for Communication-Based Design.* In Catuscia Palamidessi, editeur, CONCUR 2000 – Concurrency Theory, volume 1877 of *Lecture Notes in Computer Science*, pages 29–47. Springer Berlin Heidelberg, 2000.

[Sangiovanni-Vincentelli *et al.* 2004] A.d Sangiovanni-Vincentelli, L.a Carloni, F.a b De Bernardinis et M.c Sgroi. *Benefits and Challenges for Platform-Based Design.* In Conference of Proceedings of the 41st Design Automation Conference, pages 409–414, San Diego, CA, Juin 2004.

[Sangiovanni-Vincentelli 2007] A. Sangiovanni-Vincentelli. *Quo Vadis, SLD ? Reasoning About the Trends and Challenges of System Level Design.* Proceedings of the IEEE, vol. 95, pages 467–506, Mars 2007.

[Schwalbe 2013] Kathy Schwalbe. Information technology project management. Cengage Learning, 2013.

[Seligmann *et al.* 1989] P.S. Seligmann, GM.M. Wijers et H.G. Sol. *Analyzing the Structure of I.S. Methologies, an Alternative Approach.* In Proceedings of the First Dutch Conference on Information Systems, 1989.

[Sifakis 2011] Joseph Sifakis. *A Vision for Computer Science – the System Perspective.* Central European Journal of Computer Science, pages 108–116, Mars 2011.

[Sindico *et al.* 2012] Andrea Sindico, Marco Natale et Alberto Sangiovanni-Vincentelli. *An Industrial System Engineering Process Integrating Model Driven Architecture and Model Based Design.* In RobertB. France, Jürgen Kazmeier, Ruth Breu et Colin Atkinson, editeurs, Model Driven Engineering Languages and Systems, volume 7590 of *Lecture Notes in Computer Science*, pages 810–826. Springer Berlin Heidelberg, Octobre 2012.

[Springgay 2001] Dave Springgay. *Using Perspectives in the Eclipse UI.* `https://www.eclipse.org/articles/using-perspectives/PerspectiveArticle.html`, Août 2001.

[Strategy Analytics 2012] Strategy Analytics. *HTC, Apple & Samsung Top Smartphone Retail Price Decline Comparison*, Septembre 2012.

[Szyperski 2002] Clemens Szyperski. Component Software : Beyond Object-Oriented Programming. Pearson Education, 2002.

[Tanenbaum & Steen 2006] Andrew S. Tanenbaum et Maarten van Steen. Distributed systems : Principles and paradigms (2nd edition). Prentice-Hall, Inc., Upper Saddle River, NJ, USA, Octobre 2006.

[The Metropolis Project Team 2004] The Metropolis Project Team. *The Metropolis Meta Model*, Septembre 2004.

[Törngren 2013a] Martin. Törngren. *Discussion with professor Törngren after the "Integrating Viewpoints in the Development of CPS" presentation given at the Cyber-Physical System Summer School*, Juillet 2013.

[Törngren 2013b] Martin. Törngren. *Integrating Viewpoints in the Development of CPS*. video, Juillet 2013.

[Vanderperren *et al.* 2008] Yves Vanderperren, Wolfgang Mueller et Wim Dehaene. *UML for electronic systems design : a comprehensive overview*. Design Automation for Embedded Systems, vol. 12, no. 4, pages 261–292, Décembre 2008.

[Verimag Laboratory 2013a] Verimag Laboratory, 2013.

[Verimag Laboratory 2013b] Verimag Laboratory, 2013.

[Vickoff 2000] Jean-Pierre Vickoff. *Methode RAD–Elements fondamentaux*, 2000.

[Vidal *et al.* 2009] J. Vidal, F. de Lamotte, G. Gogniat, P. Soulard et J.-P. Diguet. *A Co-Design Approach for Embedded System Modeling and Code Generation with UML and MARTE*. In Design, Automation Test in Europe Conference Exhibition, pages 226–231, Avril 2009.

[Wolf *et al.* 2008] W. Wolf, A.A. Jerraya et G. Martin. *Multiprocessor system-on-chip (MPSoC) technology*. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 27, no. 10, pages 1701–1713, Octobre 2008. cited By (since 1996) 62.

[Wolf 2003] Wayne Wolf. *A Decade of Hardware/Software Codesign*. Computer, vol. 36, no. 4, pages 38–43, Avril 2003.

[Wood *et al.* 2008] S.K. Wood, D.H. Akehurst, O. Uzenkov, W.G.J. Howells et K.D. McDonald-Maier. *A Model-Driven Development Approach to Mapping UML State Diagrams to Synthesizable VHDL*. Computers, IEEE Transactions on, vol. 57, no. 10, pages 1357–1371, Oct 2008.

[Zoughbi *et al.* 2011] Gregory Zoughbi, Lionel Briand et Yvan Labiche. *Modeling Safety and Airworthiness (RTCA DO-178B) Information – Conceptual Model and UML Profile*, Juin 2011.

Résumé

Le développement des systèmes embarqués est complexe. Cette complexité a plusieurs origines. D'une part, elle provient des caractéristiques propres des systèmes embarqués (mesure et contrôle du monde physique, exécution sur une plate-forme physique limitée en ressources, autonomie, fiabilité, réactivité, ...) qui les distinguent des systèmes purement logiciels. D'autre part, elle est due aux fortes contraintes industrielles auxquelles ces systèmes sont soumis : coûts et délais de développement et de fabrication, équipes pluri-disciplinaires, certification et documentation des systèmes. Afin de maitriser cette complexité, un certain nombre de méthodes et de langages furent proposés. Ils mettent l'accent sur une modélisation de l'application et de la plate-forme constituant le système embarqué. Cependant, les notions de méthode et de processus de développement qui abordent le problème de description des activités à réaliser ne sont pas bien connues dans le domaine de l'ingénierie des systèmes embarqués et les méthodes actuelles tirent peu parti de l'expérience capitalisée dans d'autres domaines d'ingénierie tels que les systèmes d'information. L'enjeu de cette thèse est la définition, la formalisation et l'outillage d'une méthode couvrant le développement des systèmes embarqués. Pour ce faire, ces travaux ont été axés autour de quatre contributions majeures : (1) la formalisation d'un processus guidé et d'un langage permettant une modélisation homogène d'une application et de sa plate-forme, (2) la composition de plates-formes complexes permettant une implémentation progressive d'une application sur sa plate-forme réelle, (3) l'intégration de la gestion de projet et de la traçabilité couplées aux produits offrant au chef de projet un moyen de mesurer et de piloter l'avancement de progression, d'organiser son équipe et de paralléliser les développements, et (4) le développement d'un outil dédié aux supports du processus, du langage et de la gestion de projet.

Abstract

Embedded system development is complex. This complexity has several sources. A first one is embedded system own specificities (physical world measurement and control, execution on a physical resource-constrained platform, reliability, responsiveness, ...) that distinguish themselves from software systems. Another one comes from industrial concerns about whom these systems are subject to : product and development costs and delays, multidisciplinary teams, system documentation and certification. To handle this complexity, few methods and languages have been proposed. They focus on a modeling of both application and platform part included in an embedded system. However, the notions of method and process model are barely known from the embedded system community and current methods do not capitalize on the knowledge acquired by other engineering domains like information systems. The goal of this thesis is the definition, the formalization and the tooling of an embedded system development method. To do that, this work focuses on four main contributions : (1) the formalization of a guided process and a language to ensure a consistent modeling of both the application and the platform, (2) the composition of complex platforms to permit a progressive implementation of an application on its concrete platform, (3) the integration of a project management and a product traceability allowing the project manager to measure and monitor the development progress, to organize his team and to parallelize the development, and (4) the development of a tool designed to support the process, the language and the project management.