

Modélisation et analyse de systèmes embarqués

Fabrice KORDON, Jérôme HUGUES, Agusti CANALS, et Alain DOHET

7 novembre 2012

Table des matières

Préface	13
Dominique POTIER	
Chapitre 1. Introduction générale	15
Fabrice KORDON, Jérôme HUGUES, Agusti CANALS et Alain DOHET	
1.1. Problématique	15
1.2. Objectif du présent ouvrage	16
1.3. Plan de l'ouvrage	17
1.4. Remerciements	17
PREMIÈRE PARTIE. PROLÉGOMÈNES	19
Chapitre 2. Eléments pour la conception de systèmes embarqués	21
Fabrice KORDON, Jérôme HUGUES, Agusti CANALS et Alain DOHET	
2.1. Introduction	21
2.2. Modélisation de systèmes	23
2.3. UML en bref	24
2.4. Approches de développement dirigée par les modèles	28
2.5. Analyse des systèmes	32
2.6. Aspects méthodologiques du développement de systèmes embarqués .	38
2.7. Conclusion	42
2.8. Bibliographie	42
Chapitre 3. Etude de cas : le pacemaker	45
Fabrice KORDON, Jérôme HUGUES, Agusti CANALS et Alain DOHET	
3.1. Introduction	45
3.2. Le cœur, le pacemaker	46
3.3. Spécification du cas d'étude	49

3.4. Conclusion	58
3.5. Bibliographie	59
DEUXIÈME PARTIE. SysML	61
Chapitre 4. Présentation des concepts de SysML	63
Jean-Michel BRUEL et Pascal ROQUES	
4.1. Introduction	63
4.2. Origines de SysML	64
4.3. Présentation générale : les neuf types de diagrammes	64
4.4. Modélisation des exigences	66
4.5. Modélisation structurelle	69
4.6. Modélisation dynamique	76
4.7. Modélisation transverse	82
4.8. Environnement et outillage	85
4.9. Conclusion	85
4.10. Bibliographie	85
Chapitre 5. Modélisation de l'étude de cas avec SysML	87
Loïc FEJOZ, Philippe LEBLANC et Agusti CANALS	
5.1. Introduction	87
5.2. Spécification du système	89
5.3. Conception du système	96
5.4. Traçabilité et allocations	106
5.5. Modèle de test	108
5.6. Conclusion	111
5.7. Bibliographie	112
Chapitre 6. Analyse des exigences	113
Ludovic APVRILLE et Pierre DE SAQUI-SANNES	
6.1. Introduction	113
6.2. Le langage AVATAR et l'outil TTool	114
6.3. Expression AVATAR du modèle SysML de pacemaker enrichi	117
6.4. Architecture	118
6.5. Comportement	120
6.6. Vérification formelle du mode VVI	121
6.7. Positionnement par rapport à d'autres travaux	127
6.8. Conclusion	129
6.9. Pour aller plus loin	129
6.10. Bibliographie	130
TROISIÈME PARTIE. MARTE	133

Chapitre 7. Présentation des concepts de MARTE	135
Sébastien GÉRARD et François TERRIER	
7.1. Introduction	135
7.2. Généralités	135
7.3. Quelques détails de MARTE	142
7.4. Conclusion	152
7.5. Bibliographie	152
Chapitre 8. Modélisation de l'étude de cas avec MARTE	153
Jérôme DELATOUR et Joël CHAMPEAU	
8.1. Introduction	153
8.2. Spécification logicielle	155
8.3. Conception logicielle système – partie architecturale	159
8.4. Conception logicielle système – partie comportementale	165
8.5. Conclusion	169
8.6. Bibliographie	170
Chapitre 9. Analyse à partir du modèle	171
Frédéric BONIOL, Philippe DHAUSSY, Luka LE ROUX et Jean-Charles ROGER	
9.1. Introduction	171
9.2. Modèle à vérifier et exigences	175
9.3. <i>Model-checking</i> des exigences	180
9.4. Exploitation des contextes	187
9.5. Bilan	195
9.6. Conclusion	196
9.7. Bibliographie	196
Chapitre 10. Déploiement et génération de code à partir du modèle	199
Chokri MRAIDHA, Ansgar RADERMACHER et Sébastien GÉRARD	
10.1. Introduction	199
10.2. Les modèles d'entrée	201
10.3. Génération du modèle d'implantation	204
10.4. Génération de code	212
10.5. Support outillé	215
10.6. Conclusion	216
10.7. Bibliographie	217
QUATRIÈME PARTIE. AADL	219
Chapitre 11. Présentation des concepts de AADL	221
Jérôme HUGUES et Xavier RENAULT	
11.1. Introduction	221

12 Systèmes embarqués

11.2. Concepts généraux des ADL	221
11.3. AADLv2, un ADL pour la conception et l'analyse	222
11.4. Taxonomie des entités AADL	225
11.5. Annexes à AADL	233
11.6. Analyses de modèles AADL	235
11.7. Conclusion	237
11.8. Bibliographie	238
Chapitre 12. Modélisation de l'étude de cas avec AADL	241
Etienne BORDE	
12.1. Introduction	241
12.2. Rappels concernant la structure du pacemaker	243
12.3. Modélisation AADL de la structure du pacemaker	244
12.4. Rappels concernant le fonctionnement du pacemaker	249
12.5. Modélisation AADL de l'architecture logicielle du générateur de pulsations	254
12.6. Modélisation du déploiement des fonctionnalités du pacemaker	263
12.7. Conclusion	264
12.8. Bibliographie	265
Chapitre 13. Analyse à partir du modèle	267
Thomas ROBERT et Jérôme HUGUES	
13.1. Introduction	267
13.2. Validation comportementale par mode et globale	268
13.3. Conclusion	278
13.4. Bibliographie	279
Chapitre 14. Génération de code à partir du modèle	281
Laurent PAUTET et Bechir ZALILA	
14.1. Introduction	281
14.2. Génération de composants applicatifs	284
14.3. Génération de composants intergiciels	299
14.4. Déploiement et configuration des composants intergiciels	300
14.5. Intégration de la chaîne de compilation	302
14.6. Conclusion	303
14.7. Bibliographie	304

Préface

En introduction du rapport¹ de la mission « Briques génériques du logiciel embarqué » qui m'avait été confiée en 2010 par le ministre chargé de l'industrie, la secrétaire d'Etat à la prospective et au développement de l'économie numérique et le Commissaire général aux investissements d'avenir, il est rappelé que :

« Les technologies des systèmes embarqués, logiciel embarqué et microélectronique, ont la capacité de transformer tous les objets du monde physique – du plus petit au plus grand, du plus simple au plus complexe – en objets numériques, intelligents, autonomes et communicants. L'émergence du Web des Objets, jonction du monde du Web et de celui des systèmes embarqués, amplifie de façon considérable cette révolution. »

De fait, le déploiement généralisé des systèmes embarqués modifie profondément notre environnement, est porteur de très nombreuses innovations de produits et d'usages et impacte l'ensemble des activités industrielles et de services.

La maîtrise de l'ingénierie des systèmes embarqués constitue donc un élément-clé de compétitivité industrielle. Toutefois, atteindre cette maîtrise est encore une démarche complexe et coûteuse en raison à la fois de la largeur du champ technologique à couvrir, de la diversité et de la complexité des exigences et des solutions. Cela est bien illustré par les différentes « vues » de cette ingénierie proposées par la *Embedded Systems Common Technical Baseline*² ou CTB :

Préface rédigée par Dominique POTIER.

1. Briques génériques du logiciel embarqué, Dominique Potier, octobre 2010 (voir www.ladocumentationfrançaise.fr/rapports-publics).

2. Développée en 2008 par B. Bouyssounouse (VERIMAG), T. Veitshans et C. Lecluse (CEA), la *embedded systems common technical baseline* constitue à la fois un référentiel, un guide et une base de connaissance en matière de systèmes embarqués (voir www.embedded-systems-portal.com/CTB).

14 Systèmes embarqués

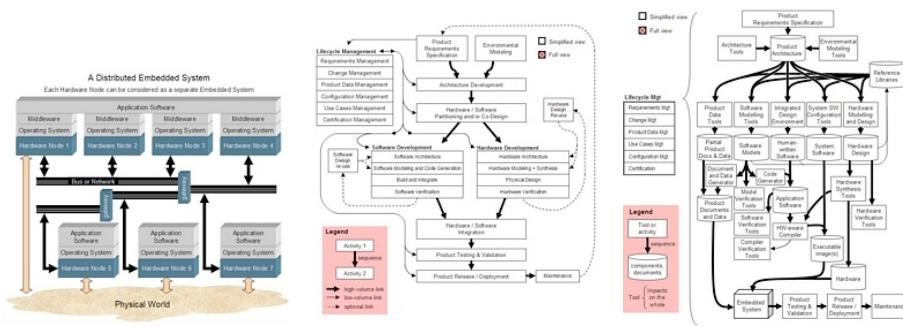


Figure 1 – Vues « système » (à gauche), « cycle de vie » (au centre) et « outils de conception » (à droite)

- la vue « Système » décrit le système embarqué proprement dit, c'est-à-dire son architecture et l'ensemble de ses composants matériels et logiciels (calculateurs, bus et réseaux OS, *middleware*, logiciels applicatifs) ;
- la vue « Outils de conception » montre l'ensemble des outils de conception (modélisation, simulation) utilisés ;
- la vue « Cycle de vie » du produit décrit les activités et leur séquencement qui interviennent dans le cycle industriel de conception, développement et maintenance d'un système embarqué.

L'examen de ces différentes vues met en évidence l'importance des activités de modélisation dans l'ingénierie des systèmes embarqués. Ainsi, à chaque niveau et pour chaque élément de la vue « Système » – architecture, processeurs, bus et réseaux, logiciels, etc. – correspondent, comme figuré dans les vues « Cycle de vie » du produit et « Outils de conception », des modèles et des outils de modélisation permettant d'assister et de valider les différentes étapes de développement.

La nouvelle monographie de la SEE « Modélisation et analyse de systèmes embarqués » est donc particulièrement bienvenue. Elle répond à une demande tant des ingénieurs que des étudiants de disposer d'un ouvrage abordable, présentant les formalismes et les principes de la modélisation des systèmes embarqués. Il faut noter à ce propos qu'il y a une attente particulièrement forte de la part de nombreux ingénieurs de formation initiale en électronique qui doivent faire face à une évolution rapide de leurs activités vers la conception et la modélisation système et logiciel. Dans le rapport cité, il est mentionné que pour les PME et les bureaux d'études électroniques, la mutation vers le logiciel embarqué est encore largement à faire. Comme le constate le programme CAP'TRONIC (Compétitivité et Innovation des PME par l'électronique), ces entreprises disposent aujourd'hui de trop peu de compétences en développement de logiciels embarqués (manque de méthodologies et d'outils) et sont de plus en plus

confrontés à des exigences fortes de leurs clients (donneurs d'ordre, intégrateurs, utilisateurs finaux) en termes de robustesse et de sûreté de fonctionnement.

Un autre intérêt de l'ouvrage est la démarche suivie pour l'exposé du sujet. De façon tout à fait pédagogique, la présentation de chaque méthode de modélisation est suivie d'une mise en œuvre détaillée sur une étude de cas commune à travers laquelle les étapes et activités décrites dans les vues présentées plus haut sont très clairement démontrées.

Enfin, cet ouvrage illustre le dynamisme et la qualité de la communauté de recherche en France, tant académique qu'industrielle, en matière de langages et d'outils de modélisation de systèmes embarqués ainsi que son rôle dans les instances de standardisation de ces langages. Il faut à cette occasion redire l'importance de ces activités de standardisation qui permettent que des avancées scientifiques et technologiques soient reconnues et validées par tous au niveau international et constituent la référence pour les développeurs d'outils. Ainsi, le standard MARTE (*UML Profile for Modeling and Analysis of Real-Time Embedded Systems*) présenté dans la troisième partie du livre résulte très directement d'une initiative lancée en 2006 par un groupe de chercheurs de THALES et du CEA LIST (dont plusieurs auteurs du livre) et qui a abouti en 2009 à l'adoption par l'OMG (*Object Management Group*) du standard MARTE 1.0. De même, plusieurs des auteurs des deuxièmes et quatrièmes parties du livre sont également actifs dans les instances de normalisation de SysML et d'ADDL.

Je souhaite que ce livre rencontre le meilleur succès et qu'il contribue à former la prochaine génération de concepteurs de systèmes embarqués.

Chapitre 1

Introduction générale

1.1. Problématique

Depuis la construction du premier d'entre eux dans les années 1960, l'*Apollo Guidance Computer* [WIK 12a], les systèmes embarqués n'ont cessé de se diffuser. Ils rendent un nombre de services sans cesse grandissant et font partie de notre vie quotidienne : ascenseurs, transports (signalisation, avions, automobiles, trains), téléphonie, médecine, énergie, industrie, etc.

Ainsi, si l'on parle de plus en plus de systèmes embarqués, on oublie en général quelles sont leurs caractéristiques premières. La définition qu'en donne Wikipedia est la suivante [WIK 12c] :

« On désigne sous le terme informatique embarqué les aspects logiciels se trouvant à l'intérieur des équipements n'ayant pas une vocation purement informatique. L'ensemble logiciel, matériel intégré dans un équipement constitue un système embarqué. »

Ainsi, un point central de leur développement est leur interaction avec leur environnement, et l'impact qui peut en résulter en termes de sécurité et de fiabilité.

Le développement de ces systèmes est un problème difficile qui n'a pour l'instant pas de solution globale. Certes, des avions volent, des véhicules robotisés se rendent sur

Introduction rédigée par Fabrice KORDON, Jérôme HUGUES, Agusti CANALS et Alain DOHET.

des planètes à des centaines de millions de kilomètres de la Terre. Mais le développement de systèmes embarqués dans ces missions nécessite encore une énergie et un travail colossal, impliquant la mise en œuvre de technologies complexes et difficiles à maîtriser.

Un autre difficulté est que ces systèmes sont plongés dans le monde réel, qui n'est pas discrétilisé (comme l'appréhende en général l'informatique) et dont le comportement est d'une richesse qui empêche parfois toute hypothèse simplificatrice.

Enfin, ils sont en général autonomes et doivent faire face à d'éventuelles situations imprévues (incidents par exemple), voire à des situations hors des hypothèses de conception initiales. A titre d'exemple, considérons les satellites de communication SPOT [WIK 12b]. Le projet a démarré en 1978 et les satellites lancés à partir de 1986, les derniers (sixième génération) devant être lancés en 2012 et 2013. SPOT-2, originellement prévu pour durer trois ans, a fonctionné pendant vingt ans. Cela illustre l'impossibilité, au moment de la conception, de décider des usages potentiels un quart de siècle plus tard¹.

1.2. Objectif du présent ouvrage

Au départ « artisanale », la construction de ces systèmes s'industrialise à mesure que la diversité des applications et de leur usage croît. Des méthodologies s'ébauchent, se perfectionnent, se raffinent.

L'objectif de cet ouvrage est de faire le point, sur l'état de l'art du développement des systèmes embarqués et, en particulier, se concentrer sur leur modélisation et leur analyse. Il s'agit d'opérations cruciales qui détermineront la fiabilité du futur système.

A l'heure actuelle, il existe trois approches récentes, dans la lignée des technologies d'« Ingénierie Dirigée par les Modèles » (IDM²) : SysML, UML/MARTE et AADL. Elles sont toutes les trois le résultat d'un travail collaboratif international et visent différents aspects des systèmes embarqués. SysML se focalise sur l'aspect ingénierie système tandis qu'UML/MARTE et AADL visent plutôt, dans des contextes différents et avec des approches différentes, la phase de conception, d'analyse et de développement.

Il nous a donc semblé important de les évoquer toutes les trois dans cet ouvrage afin de donner au lecteur une vision globale de leurs possibilités. Nous démontrons également les apports de chacune d'elles à différents stades du cycle de vie du logiciel.

1. Rappelons que l'informatique est une science qui a un peu plus de 70 ans si l'on se réfère à la création du premier ordinateur, le Z3 [WIK 12d] par Konrad Zuze en 1941.

2. La section 2.4 (chapitre 2) est consacrée à cette notion.

1.3. Plan de l'ouvrage

Cet ouvrage est structuré en quatre parties.

La première partie comporte deux chapitres. Le chapitre 2 présente des concepts généraux qui nous ont paru utiles. Comme deux des langages de spécification abordées s'appuient sur UML, il nous a semblé utile de présenter cette notation. De même, nous faisons un point sur les principaux enjeux du développement de systèmes embarqués.

Le chapitre 3 présente le cahier des charges simplifié d'une étude de cas commune aux approches abordées dans cet ouvrage : un *pacemaker*. Il s'agit d'une version simplifiée d'une spécification publiée en 2007 par *Boston Scientific* [Bos 07]. Ce cas d'étude constitue un fil conducteur permettant au lecteur d'observer comment les différents aspects d'un même système sont abordés dans les différentes approches. L'intérêt de cette étude est qu'elle a également été abordée avec d'autres méthodes que celles présentées dans cet ouvrage, ce qui permet de comparer leur pouvoir d'expression et les outils d'analyses associés.

Les parties suivantes abordent chacune une des approches sélectionnées dans cet ouvrage : SysML (deuxième partie), MARTE (troisième partie) et AADL (quatrième partie). Elles respectent toutes la même structure à quelques détails près. Le premier chapitre présente de manière synthétique la notation associée à la démarche et le second expose la modélisation de l'étude de cas. Suit ensuite un chapitre dédié à l'analyse de la spécification. Dans les troisième et quatrième parties, le lecteur trouvera un quatrième chapitre dédié à la génération de code.

1.4. Remerciements

Avant de clore cette introduction, les coordinateurs souhaitent remercier les différents acteurs sans qui cet ouvrage n'existerait pas :

- Dominique Potier, du pôle System@tic qui a rédigé la préface de cet ouvrage ;
- les nombreux auteurs, qui, dispersés sur le territoire national, ont su se synchroniser et échanger leur points de vue pour présenter une vue cohérente des approches considérées ;
- la SEE et son comité éditorial qui nous a encouragé à poursuivre l'effort de rédaction ;
- le GDR GPL (Génie de la Programmation et du Logiciel [CNR 12]) du CNRS qui, par sa subvention, a permis l'organisation de réunions de synchronisation des auteurs lors de la préparation de cet ouvrage.

Bibliographie

18 Systèmes embarqués

- [Bos 07] BOSTON SCIENTIFIC, PACEMAKER System Specification, janvier 2007.
- [CNR 12] CNRS, « Page d'accueil du GDR GPL », gdr-gpl.cnrs.fr, 2012.
- [WIK 12a] WIKIPÉDIA, « Apollo Guidance Computer », fr.wikipedia.org/wiki/Apollo_Guidance_Computer, 2012.
- [WIK 12b] WIKIPÉDIA, « SPOT (satellite) », [fr.wikipedia.org/wiki/SPOT_\(satellite\)](http://fr.wikipedia.org/wiki/SPOT_(satellite)), 2012.
- [WIK 12c] WIKIPÉDIA, « Système embarqué », fr.wikipedia.org/wiki/Système_embarqué, 2012.
- [WIK 12d] WIKIPÉDIA, « Zuse 3 », fr.wikipedia.org/wiki/Zuse_3, 2012.

PREMIÈRE PARTIE
Prolégomènes

Chapitre 2

Eléments pour la conception de systèmes embarqués

2.1. Introduction

Le développement de systèmes embarqués se fait traditionnellement avec des langages de bas niveaux : langages d'assemblages puis souvent, C. La raison principale en est le besoin d'élaborer des programmes aux empreintes mémoires faibles car déployés sur des architectures très compactes¹. Progressivement, des techniques de développement coûteuses se sont mises en place, principalement basées sur une analyse fine des besoins, des tests intensifs et sur des procédures de développement très strictes permettant de garantir un niveau de sécurité conforme à l'attente du grand public.

Mais ces systèmes, assurant souvent des missions critiques, avaient un type de développement coûtant fort cher, ce qui les confinait à un usage spécifique comme le spatial, l'aéronautique, le nucléaire ou le ferroviaire. Avec l'émergence de ces systèmes dans d'autres domaines d'application plus « grand public », les approches de développement ont dû évoluer dans un sens où les coûts doivent être réduits. L'évolution économique vers du développement *offshore* ne facilite pas non plus ces aspects fiabilité/sécurité mais, ces nouvelles approches de développement pourraient, à terme, devenir des solutions concurrentielles.

Chapitre rédigé par Fabrice KORDON, Jérôme HUGUES, Agusti CANALS et Alain DOHET.

1. A titre d'exemple, la sonde Pioneer 10, lancée en 1972, disposait de 6 Koctets de mémoire seulement [FIM 74], ce qui ne l'a pas empêché de transmettre une quantité impressionnante de données scientifiques concernant le système de Jupiter avant de devenir le premier objet construit par l'homme à quitter le système solaire.

Le simple usage de langages de programmation de haut niveau ne constitue pas à lui seul une solution. En effet, les systèmes embarqués contiennent souvent des mécanismes sophistiqués et les exécutifs qui vont avec. Comme ces derniers ne peuvent être certifiés, leur usage est limité. Citons à ce titre l'exemple du langage Ada qui a été élaboré dans les années 1970 avec, comme objectif, de permettre le développement de systèmes embarqués à moindre coût. Certaines de ses possibilités (typiquement la gestion du parallélisme) n'ont été utilisées qu'avec parcimonie dans certains domaines, précisément parce qu'il fallait associer le code compilé à des exécutifs réduits (par exemple, ne supportant pas le parallélisme ou l'allocation dynamique de mémoire).

L'ingénierie dirigée par les modèles (IDM ou *MDE* en anglais pour *Model Driven Engineering*) apparaît comme une solution potentielle au problème du coût de la fiabilité logicielle des systèmes embarqués. En effet, elle permet de mieux associer les différents langages et modèles utilisés tout au long du cycle de conceptions/développement.

En particulier, elle permet de s'appuyer sur des « modèles » dédiés, sur lesquels il est parfois possible de raisonner. Ainsi, les ingénieurs peuvent prédirer certains aspects du comportement de leur programmes. Les techniques de transformation qui ont été développées par cette communauté assurent le lien entre les différentes phases du développement.

Bien sûr, ces nouvelles approches ne peuvent se substituer brutalement aux démarches existantes. Les acteurs du développement de systèmes embarqués (en particulier quand ils assurent des missions critiques) ne peuvent pas prendre de risques. Mais, progressivement, ces acteurs s'intéressent à ces nouvelles techniques et à leurs applications dans un cadre méthodologique adéquat.

L'objectif est, à terme, d'aboutir à des processus industriels performants permettant de produire des logiciels de qualité à des prix concurrentiels, en permettant par exemple de faire de la co-simulation, de la vérification formelle et de la génération de code cible validée dès la modélisation.

Ce n'est pas par hasard que les trois notations présentées dans cet ouvrage – SysML, UML/MARTE et AADL – s'appuient intensivement sur l'ingénierie des modèles. C'est clairement le « sujet chaud » de la communauté actuellement.

Plan du chapitre. L'objectif de ce chapitre est d'aborder quelques éléments qui nous ont paru importants dans le développement de systèmes embarqués dans le contexte que nous venons de brosser. Nous nous intéressons en particulier :

- à l'activité de modélisation (section 2.2) ;
- à une brève présentation d'UML, qui sert de base à deux des trois notations présentées dans cet ouvrage (section 2.3) ;

- à une présentation de l'ingénierie dirigée par les modèles (section 2.4);
- à un panorama des techniques d'analyse et en particulier celles utilisées dans cet ouvrage (section 2.5);
- aux aspects méthodologiques du développement de systèmes (section 2.6).

2.2. Modélisation de systèmes

L'opération de modélisation est pratiquée par l'humanité depuis toujours. Elle vise à expliquer le comportement d'un système complexe au moyen d'un modèle qui l'abstrait. Pour s'en convaincre, évoquons les différents modèles imaginés pour expliquer le mouvement céleste : système des épicycles, système de Ptolémé, système de Tycho Brahe, système de Copernic, etc. Leur objectif était de prédire l'évolution de la position des planètes. Ils sont le raffinement successif d'une meilleure compréhension du domaine, un nouveau système remplaçant le précédent lorsque l'évidence est faite qu'icelui ne correspond pas à la réalité. La modélisation est donc, depuis longtemps un outil indispensable des sciences expérimentales.

En informatique, la grande différence réside dans le fait que le modèle ne reproduit pas un système que l'on observe pour en comprendre le comportement. Il se situe en amont et permet d'identifier qu'une solution en cours d'élaboration respectera les propriétés attendues.

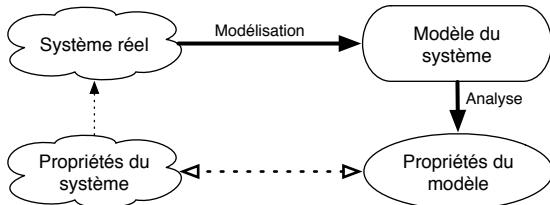


Figure 2.1 – Relations entre le système, son modèle et leurs propriétés

La figure 2.1 représente les relations entre les différentes entités d'un système au sein de l'opération de modélisation. L'ingénieur modélise le système qu'il est en train de concevoir. L'analyse de ce modèle (par simulation ou par le biais de méthodes formelles, voir section 2.5.1) lui permet d'en déduire des propriétés. Mais la relation entre les propriétés du modèle et les propriétés du système ne sont pas triviales.

Tout d'abord, l'expression des propriétés peut être très différente dans les deux cas. Typiquement la propriété du système sera exprimée en langage naturel alors que la propriété sur le modèle pourra avoir la forme d'un invariant ou référencera de manière

précise une entité de la spécification dont l’usager doit connaître la signification s’il veut la comprendre. Il faut donc prévoir des éléments de traçabilité entre les deux.

Ensuite, la nécessaire abstraction de certains éléments du système lors de l’opération de modélisation (par exemple, le passage du continu au discret) peut, lorsqu’elle est mal maîtrisée, engendrer un modèle qui ne représente pas exactement le système. Si c’est un sur-ensemble, alors une propriété non vérifiée sur le modèle peut l’être sur le système, si c’est un sous-ensemble c’est une propriété vérifiée sur le modèle qui peut ne pas l’être sur le système.

La modélisation est donc une opération délicate qui doit être menée avec beaucoup de précautions. En particulier, les choix de modélisation doivent être correctement documentés. Dans le domaine qui nous intéresse, les systèmes embarqués, nous introduisons plusieurs notions importantes concernant la notion de modèle.

Modélisation structurelle ou comportementale. La première vise à définir la structure d’un système (c’est-à-dire les diagrammes de classes ou d’instances en UML) tandis que la seconde décrit son comportement. Pour une recherche de comportements à problèmes, ou pour identifier des comportements émergents, il faut atteindre le second niveau qui suppose en général l’existence préalable de la modélisation de la structure du système (au moins l’identification des acteurs qui interagissent).

Modèle ouvert ou fermé. Un modèle ouvert décrit un système. Un modèle fermé décrit le système et l’environnement avec lequel il interagit. On peut ainsi considérer que le modèle ouvert est « inclus » dans un modèle fermé.

L’intérêt de distinguer les deux est que l’on peut ainsi soumettre le modèle d’un système à différentes conditions d’exécution. Chacune d’entre elles est alors caractérisée par une spécification dédiée de l’environnement qui vient « fermer » la spécification. Cette notion est tout particulièrement utile lorsque l’on s’intéresse à différents modes d’exécution : mode nominal, modes dégradés suivant certaines conditions, etc.

Notations vues dans cet ouvrage. Cet ouvrage s’intéresse aux systèmes embarqués et n’évoque donc que les notations permettant de décrire ces systèmes. Nous en avons sélectionné trois. SysML [OMG 12a] et MARTE [OMG 12b] sont des profils d’UML. AADL [SAE 09], quant à lui, est une notation dédiée intégrant à la fois la description de la partie logicielle et celle de la partie matérielle d’un système.

2.3. UML en bref

La première version d’UML était la version 2.8 de la « Méthode Unifiée » écrite par G. Booch et J. Rumbaugh. A l’époque, en 1995, Y. Jacobson ne faisait pas encore partie de l’aventure, et le « M » signifiait « Method ».

Un an plus tard, furent publiées les premières versions d'UML, la 0.9 et la 0.91 qui intégraient les travaux d'Y. Jacobson. Cette fois le « M » signifiait « Modelling ». En effet ces trois « fabuleux évangélistes », ne s'étant pas entendus sur une méthode standard, s'étaient concentrés sur la notation.

UML 1.0 et 1.1 furent proposées à l'OMG en 1997. Il y eut ensuite la version 1.2 en 1998, la 1.3 (révision très significative) en 1999, la 1.4 en 2001 et la 1.5 en 2002. Cette suite de versions est traditionnellement appelée « UML1.X ». Puis vint l'avènement d'« UML 2.X », dont la première version stable (2.0), apparut en 2003. C'est une révision majeure qui passe de neuf à treize diagrammes. Le métamodèle subit également de profondes modifications (qui impactent essentiellement les concepteurs d'outils). La notion de profil est formalisée, ce qui a permis par exemple la naissance de « variantes » comme SysML et MARTE que nous détaillons dans cet ouvrage.

Notons que le passage de 1.X à 2.X, qui était sensé améliorer l'utilisation et l'efficacité du langage, a donné des résultats en demi-teinte. En effet, l'industrie et les outilleurs avaient mis un certain temps à supporter complètement la génération 1.X et la nécessaire adaptation a retardé l'utilisation opérationnelle d'UML 2.X dont l'utilisation efficace commence juste dans l'industrie.

Nous présentons les principales caractéristiques d'UML (dans sa version courante en septembre 2012, la 2.4.1). Les treize diagrammes d'UML sont divisés en deux classes : les diagrammes statiques et les diagrammes dynamiques.

Parmi ces treize diagrammes, quatre sont considérés comme « principaux » : *classes*, *sequence*, *use cases*, *state machines*. Ils sont utilisés dans les parties consacrées à SysML et à MARTE mais aussi, de manière systématique, par tous les utilisateurs d'UML.

Nous insistons sur le fait qu'UML est une notation. Pour bien l'utiliser, il faut également une méthode. Ainsi, chaque société a créé sa propre méthode en s'appuyant le plus souvent sur la « méthode unifiée » (UM) ou sur le *Rational Unified Process* (RUP) (les deux sont assez proches). Ces méthodes décrivent quand et comment utiliser tel ou tel diagramme, comment organiser le modèle ainsi que les documents et les codes qui sont générés tout en respectant les processus de spécification/conception en vigueur dans l'entreprise comme la traçabilité, la gestion de configuration, la qualité (règles de bon usage, de codage, etc.).

2.3.1. *Les diagrammes statiques d'UML*

Ils décrivent les aspects statiques d'un système (relatifs à leur organisation). On trouve dans cette classe les diagrammes suivants : *Classes*, *Composites*, *Components*, *Déploiement*, *Objet*, *Package*.

Diagramme de classes. Ils permettent de modéliser le domaine (pour simplifier, les données et/ou les concepts manipulés par l’application), puis l’application elle-même. Nous aurons donc les classes du domaine, puis les classes applicatives (d’analyse et de conception). Un modèle moyen fait en général apparaître une centaine de classes.

Diagramme composite. Ils permettent de décomposer une classe (en général une classe applicative) en parties. Cette décomposition permet de montrer, par exemple, quelle partie implémente quelle opération, ou comment communiquent les différentes parties.

Notons qu’il existe un deuxième type de diagramme composite, il permet de modéliser des *patterns*. Une fois modélisés ces derniers peuvent être utilisés dans les diagrammes de classes.

Diagramme de composants. Ils permettent de décrire un système à travers des composants et les interactions entre composants au moyen de leurs interfaces. Un composant peut également se décomposer en sous-composants (comme les classes en parties) à travers un diagramme composite.

La différence entre une classe et un composant fait l’objet d’un vaste débat. Notre point de vue (il en existe d’autres) se résume comme suit :

- une classe est la brique de base d’un logiciel ;
- un composant est également une brique de base mais d’un niveau d’abstraction différent ; il regroupe en général un ensemble de classes, mais aussi d’autres artefacts comme des fichiers de configuration ;
- les classes sont liées entre elles par des relations (héritage, association, composition) et proposent des interfaces ;
- les composants sont liés entre eux à travers les interfaces ;
- les classes peuvent se décomposer en parties ;
- les composants peuvent se décomposer en composants seulement (de plus petite granularité).

Diagramme de déploiement. Ils permettent de modéliser l’architecture physique (composants physiques) de l’application : les machines, les processus, les modes de communication, etc. En général, un processus est composé de composants (qui sont composés de classes et autres artefacts) et distribué sur une à n machines.

Diagramme d’objets. Ils instantient un diagramme de classes. Un diagramme de classes est un modèle générique (par exemple, la description de l’organisation d’une entreprise) alors qu’un diagramme d’objets est un modèle spécifique (par exemple, la description de l’organisation d’une entreprise).

En général, on démarre par les diagrammes de classes et on s'assure de leur pertinence avec les diagrammes d'objets. Certains concepteurs préfèrent cependant démarrer des objets (du spécifique) et généraliser ensuite pour trouver les classes. Le résultat final est souvent similaire.

Diagramme de package. Ils permettent d'organiser le modèle. Un *package* peut contenir des *packages*, des classes, des objets, des diagrammes. En général, un modèle est organisé en trois *packages* : analyse des besoins (les *use cases*), architecture logique (les classes) et architecture physique (les machines, processus, composants). Chacun de ces *packages* peut ensuite être décomposé en fonction de la méthode appliquée dans l'entreprise.

Il est important de noter qu'un *package* n'est pas un composant (logique ou physique) mais une unité de structuration du modèle.

2.3.2. *Les diagrammes dynamiques d'UML*

Ils décrivent les aspects dynamiques d'un système (i.e. relatifs à leur exécution). On trouve dans cette catégorie les diagrammes suivants : Activité, Interactions (*Sequence*, *Communication*, *Overview*, *Timing*), *Use Cases*, Machines à états.

Diagramme d'activité. Ce type de diagramme (qui inclut les concepts de parallélisme) permet de modéliser un algorithme par un enchaînement d'activités/actions. Il propose notamment un langage d'actions très évolué permettant de spécifier en détail tout traitement algorithmique.

Diagrammes d'interactions (*sequence*, *communication*, *overview*, *timing*). Les diagrammes de séquence et de communication permettent de modéliser la collaboration entre les objets (instances de classes). Ces deux diagrammes sont quasiment équivalents même s'ils proposent des spécificités. La principale différence est visuelle, le diagramme de séquence montre une vue séquentielle alors que le diagramme de communication montre une vue spatiale.

Le diagramme d'*overview* permet de montrer un enchaînement de diagrammes sous une forme algorithmique semblable au diagramme d'activités. Cependant, les activités et actions sont des diagrammes. Cela permet de donner une meilleure lisibilité lors de la spécification d'enchaînements complexes.

Le diagramme de *timing* (issu du génie électronique) permet de modéliser des comportements séquencés par des évènements temporels (par exemple, les contraintes temporelles entre les différents états de plusieurs objets).

Cas d'utilisation. Ils permettent de décrire le système d'un point de vue externe : les acteurs (rôles) utilisant le système ainsi que les services (*use cases*) offerts par celui-ci. Les *use cases* sont en général raffinés par des diagrammes d'activité et/ou des diagrammes de séquence complétés par du texte libre.

Machines à états. Ils modélisent le comportement d'une classe active (événements asynchrones, protocoles de communication, etc.). Ils peuvent également modéliser le comportement du système (ses différents états, les conditions de passage d'un état à un autre, etc.).

Nous appelons « classe active » une classe ayant un comportement autonome.

2.4. Approches de développement dirigée par les modèles

Le début des années 2000 a vu la naissance de la version 2.0 d'UML, cette dernière a été intégrée progressivement dans les ateliers de développement, qui proposent désormais une palette d'outils de modélisation riches et nombreux, conformes à des normes synthétisant un ensemble d'expériences et d'attentes industrielles en ingénierie système réunies depuis de longues années.

Vers la même époque est apparue la notion de MDE (*Model Driven Engineering*) ou IDM en français (Ingénierie Dirigée par les Modèles).

A partir de ce concept, trois approches prépondérantes sont nées :

- l'approche de l'OMG (*Object Management Group*) : MDA (*Model Driven Architecture*), basée sur UML et MOF (le langage permettant d'écrire les métamodèles dans le monde de l'OMG) ;
- l'approche ECLIPSE : EMF (*Eclipse Modeling Framework*) basée sur ECORE (le langage permettant d'écrire les métamodèles dans le monde ECLIPSE) ;
- l'approche Microsoft : des outils et des concepts basés sur les DSL (*Domain Specific Language*).

Notons que l'approche proposée par l'OMG n'est pas outillée, contrairement aux deux autres. C'est donc à l'utilisateur de constituer son atelier.

2.4.1. *Les concepts*

L'IDM consiste principalement en l'utilisation de modèles aux différentes phases du cycle de développement d'une application. Trois niveaux sont considérés :

- les exigences ou CIM pour *Computation-Independent Model* ;
- l'analyse et la conception ou PIM pour *Platform Independent Model* ;

– l’« avant code » ou PSM pour *Platform Specific Model*.

Le principal objectif de l’IDM est l’élaboration de modèles indépendants des détails techniques des plates-formes cibles. Cette indépendance doit permettre à terme, la génération automatique (par transformation de modèles) de la grande majorité du code des applications et permettre ainsi un gain de productivité.

Un autre principe important en IDM est de permettre de transformer des modèles existants vers la représentation cible souhaitée. Ainsi, quelle que soit la technologie utilisée, il est possible de passer facilement d’une technologie à l’autre, pour peu que l’on dispose les outils de transformation nécessaires. Ces derniers sont basés sur des langages de transformation devant respecter la norme QVT (*Query/View/Transformation*) proposée par l’OMG.

2.4.2. Les technologies

Une question cruciale est de savoir quelle technologie utiliser. En effet, pour un métier donné, au moins deux choix s’offrent :

- 1) écrire un métamodèle du métier, puis réaliser un outillage adapté ;
- 2) écrire un profil UML du métier, puis utiliser un outil existant « profilé ».

Nous illustrons les avantages et inconvénients de ces deux techniques sur l’exemple de l’« éditeur de modèles ».

Dans le premier cas, il n’existe pas d’outils de modélisation pour le métier choisi. L’approche consiste en l’écriture du métamodèle du métier (un DSL), puis à générer un éditeur (en général 60 % de génération automatique avec les outils EMF). Une fois l’éditeur généré, les ingénieurs disposent d’un outil « spécifique », permettant de réaliser des modèles métier.

Dans le deuxième cas, le choix se porte sur un modeleur UML existant. L’approche consiste alors à écrire le profil du métier. Une fois ce profil créé, les ingénieurs disposent d’un outil UML « générique » profilé permettant de réaliser des modèles métier.

Dans le premier cas, la palette graphique de l’éditeur propose des concepts métier comme « buffer » ou « tâche ». Dans le deuxième cas, la palette graphique de l’éditeur propose les concepts UML habituels comme la « classe » mais cette dernière pourra être stéréotypée en un « buffer » ou une « tâche ».

Les deux techniques ont des avantages et des inconvénients. Dans le premier cas, les ingénieurs métier disposent d’un éditeur qui leur propose un vocabulaire métier, ce qui est un avantage, alors que dans le deuxième cas ils doivent passer par la notion de « classe » avant d’introduire les concepts métier, ce qui est une indirection inutile.

Pour que ces deux techniques soient équivalentes il faudrait pouvoir configurer les modéleurs UML pour qu'ils présentent les concepts métier sans passer par les concepts UML, un peu comme nous pouvons l'observer avec SysML où le fameux *block* est en fait une classe stéréotypée *block*.

Pour l'instant, les deux techniques se côtoient dans le monde de l'industrie et constituent deux chapelles qui campent sur leurs positions, chacune arguant des avantages de sa démarche.

2.4.3. Paysage du domaine

L'objectif majeur des modèles est de faciliter la compréhension mutuelle, les échanges et le travail en commun entre les acteurs d'un projet. Leur construction et leur représentation doivent donc respecter certaines conventions et règles, qui s'incarnent dans de nombreuses normes et standards.

Le langage de modélisation détermine les concepts manipulé, leur sémantique et leur représentation sous une forme textuelle ou graphique. La variété, tant des préoccupations en phase de conception/développement, que des domaines métiers, a donné lieu à tant de langages qu'il est impossible de tous les énumérer. Ils peuvent cependant être classés en trois catégories :

- les langages à vocation généraliste. Le principal est UML, SysML et MARTE pouvant être vus comme ses dérivés pour les systèmes ;
- les langages spécialisés associés à des méthodes de vérification formelle (automates, réseaux de Petri, B, Lustre, etc.) et permettant de vérifier mathématiquement certaines propriétés attendues ;
- les langages spécialisés non formels, liés à des métiers ou à des préoccupations spécifiques, dont ils intègrent par construction la terminologie, les concepts et les règles. Les DSL relèvent de cette catégorie. Des langages de *workflow*, tels que BPML (*Business Process Modeling Language*), sont utiles pour décrire les besoins d'interactivité entre les humains et les systèmes.

Dans le cas de systèmes embarqués s'intégrant dans des ensembles plus vastes, la nécessaire maîtrise d'une cohérence d'ensemble et de son évolution peut impliquer le recours à des cadres d'architecture. Ces derniers organisent et spécifient la façon de présenter l'architecture d'un système (ou l'architecture informatique d'un organisme). Pour cela, ils définissent les différents angles de vue nécessaires à sa description, ainsi que le métamodèle nécessaire pour assurer la cohérence entre les diverses vues et conduire des analyses d'impact.

Par rapport aux diagrammes prévus par des langages comme UML, les vues des cadres d'architecture se distinguent par le spectre plus large qu'elles couvrent (elles prennent

Nom du standard	Rôle du standard
<i>Concepts de base pour la modélisation</i>	
ISO/IEC/IEEE 42010 (2011) : Ingénierie des systèmes et des logiciels – Description de l’architecture	Définit les principes à respecter et les notions de base (points de vue, cadres d’architecture, langages de description des architectures, etc.)
IEEE 1471 (2000) : IEEE Recommended Practice for Architectural Description of Software-Intensive Systems	Pour mémoire : remplacé par ISO/IEC/IEEE 42010
<i>Langages de modélisation d’architectures</i>	
OMG UML : Unified Modeling Language V2	Extension d’UML pour l’ingénierie système
UML : ISO/IEC 19505 (V2.4.1 - 2012) : Technologies de l’information – Langage de modélisation unifié	Extension d’UML pour la modélisation des systèmes temps réel. Remplacé par le profil MARTE
OMG SysML : System Modeling Language Specification (V1.3 - OMG, 2012/06)	Extension d’UML pour la modélisation des systèmes temps réel et embarqués. Remplace le profil SPT
UML/SPT : Profile for Schedulability, Performance and Time (OMG, 2005)	Extension d’UML permettant la modélisation des aspects qualité de service et tolérance aux fautes des applications
UML/MARTE : Profile for Modeling and Analysis of Real-Time and Embedded Systems (MARTE 1.1 – OMG, 2011/06)	Extension d’UML avec des concepts relatifs aux tests
UML/QFTP : Profile for Modeling Quality of Service and Fault Tolerance Characteristics & Mechanisms (OMG, 2008)	Langage pour la description de l’architecture (logiciel + plate-forme d’exécution) des systèmes temps réels embarqués à performances critiques
UML Testing Profile (UTP) (V1.1 – OMG, 2012)	IDEF est une famille de langage de modélisation pour l’ingénierie des logiciels et des systèmes
SAE AS 5506 rev. A (2009) : Architecture Analysis & Design Language (AADL)	Norme de notation graphique servant à décrire les processus d’entreprise (et les interactions humains-systèmes)
IEEE Std 1320.2 (1998) - IEEE Standard for Conceptual Modeling Language - Syntax and Semantics for IDEF1X97 (IDEFobject)	Langage de description d’architecture d’entreprise
OMG BPMN 3.0 (2011) : Business Process Modeling and Notation	Cadre d’architecture pour les grands systèmes (militaires)
The Open Group - ArchiMate 1.0 Specification (2009)	Architectures informatiques d’entreprise
<i>Cadres d’architecture</i>	
NATO AC/322-D 0048 (2007) : NATO Architecture Framework V3 (NAF V3)	Description de systèmes d’information répartis
ISO 15704, INDustrial automation systems – Requirements for enterprise-reference architectures and methodologies	
The Open Group Architecture Framework (TOGAF)	
ISO/IEC 10746 (1998) : Information technology - Open distributed processing – Reference model (RM-ODP)	

Tableau 2.1 – Principales normes pour la modélisation des systèmes

en compte des dimensions nécessaires à la gouvernance d’ensemble : organisation et processus d’entreprise, évolution des capacités, synchronisation des projets, etc.) et se limitent à spécifier le type d’informations à fournir, en laissant toute liberté pour les modalités de représentation. Le NAF de l’OTAN (*NATO Architecture Framework*) est l’un des cadres d’architecture les plus récents et les plus avancés. A ce titre, il est utilisé au-delà du domaine militaire.

Le tableau 2.1 récapitule les principales normes utilisées pour la modélisation de systèmes embarqués. Les principaux organismes élaborant les normes et standards intéressants la modélisation de l’architecture des systèmes embarqués sont :

- l’ISO (*International Standard Organization*);

- l’IEEE (*Institute of Electrical and Electronics Engineers*), association professionnelle américaine ;
- l’OMG (*Object Management Group*), association américaine dont l’objectif est de standardiser et promouvoir le modèle objet sous toutes ses formes.

On peut y ajouter des acteurs « sectoriels » :

- le SAE International (*Society of Automotive Engineers*), association mondiale couvrant le domaine aérospatial et celui des véhicules terrestres, avec les standards AS (*Aerospace Standard*) et GV (*Ground Vehicle*) ;
- l’OTAN, principalement dans le domaine militaire ;
- *The Open Group*, pour les systèmes d’information d’entreprise ;
- l’IEEE (*Institute of Electrical and Electronics Engineers*), association professionnelle américaine, qui publie ses propres normes au travers de la *IEEE Standards Association* ;
- l’EIA (*Electronic Industries Alliance*), qui a cessé ses activités en 2011 et a été remplacée par cinq associations plus spécialisées, dont le GEIA (*Governmental Electronics and Information Technology Association*) ;
- l’ECSS (*European Cooperation for Space Standardization*) qui entretient un ensemble de normes dédiées à la gestion des projets spatiaux, organisé en trois branches : gestion de projet, assurance produit, ingénierie système ;
- le RTCA (*Requirements & Technical Concepts for Aviation*).

2.5. Analyse des systèmes

Depuis longtemps, les systèmes sont validés au moyen de tests et de simulations lorsque des modèles exécutables sont disponibles. Cependant, cette approche exploratoire ne garantit pas que l’ensemble des exécutions possibles sont bien couvertes. Des « zones d’ombre » dans l’exécution du système subsistent et peuvent cacher des défaillances. Cela est particulièrement vrai dans le cas de systèmes embarqués. En effet, des contraintes non fonctionnelles (par exemple, la consommation énergétique, les performances, etc.) s’ajoutent aux contraintes fonctionnelles déjà difficiles à vérifier.

Pour aller au-delà de l’« horizon » que constitue la limite des approches traditionnelles de simulation et de test, de nombreux travaux démontrent qu’il faut utiliser des méthodes formelles. Seules ces dernières garantissent qu’une propriété est vérifiée ou non car elles s’appuient sur des modèles mathématiques permettant de raisonner sur la spécification.

Dans ce domaine, il existe deux familles de techniques de vérification formelles : la preuve et le *model checking*. Nous rappelons brièvement le principe de la première famille qui ne sera pas développée dans le présent ouvrage car il n’en fera pas usage.

Nous présentons ensuite plus en détail l'approche par *model checking* qui est utilisée dans le chapitre 9.

2.5.1. Vérification formelle par preuve

Les outils et langages emblématiques de cette famille de techniques formelles sont Coq [AFF 08], Z [ISO 02] ou B [ABR 96]. Elles ont toutes leur *success stories* comme, par exemple, la vérification du comportement de la ligne 14 du métro parisien avec la méthode B.

Le principe de base est une axiomatisation du système à vérifier au moyen d'un formalisme mathématique. Les propriétés sont des théorèmes qui doivent être démontrés à partir des axiomes et, le cas échéant, *via* des lemmes et théorèmes intermédiaires. Un assistant de preuve est utilisé pour aider à explorer l'espace de la démonstration mais, hélas, n'effectue pas automatiquement cette dernière.

L'approche par preuve permet de démontrer que le système respecte les propriétés voulues. De plus, on connaît les conditions de validité de cette preuve, comme, par exemple, les paramètres de l'état initial requis pour vérifier les propriétés. On dispose donc de résultats extrêmement intéressants pour les concepteurs d'un système.

Cependant, cette technique possède quelques inconvénients. Le premier est sa difficulté de mise en œuvre. Cette approche requiert des ingénieurs hautement qualifiés maîtrisant à la fois le domaine applicatif considéré et les techniques de démonstration de théorèmes. L'autre difficulté est l'absence de diagnostic en cas d'échec de la preuve (c'est-à-dire l'exploration effectuée par l'assistant de preuve échoue). Seule une grande expertise permet, dans certains cas, de comprendre si l'échec d'une preuve est liée au système lui-même ou à une erreur de modélisation.

2.5.2. Vérification formelle par model checking

Le principe du *model checking* [QUE 82, CLA 86, CLA 00] est très simple. Le système est décrit dans une notation formelle exécutable dans laquelle un état est généralement représenté sous la forme d'un vecteur de valeurs (par exemple, l'état des variables d'un processus). Ce système est ensuite simulé de manière exhaustive, ce qui est possible car la nature d'un état permet de savoir, pour chaque nouvelle configuration explorée du système, si elle a déjà été rencontrée ou non. Ainsi, si le système est fini, on peut explorer son espace d'états de manière exhaustive et y rechercher les propriétés voulues (il existe aussi des techniques de *model checking* dédiées aux systèmes infinis [DEM 11]).

Le principal avantage de cette technique est qu'elle est complètement automatique ; son utilisation ne requiert donc pas de connaissances spécifiques de la part des ingénieurs (seul le langage de spécification doit être maîtrisé). De plus, dans la plupart

des cas, la réponse est soit « oui, la propriété est vérifiée », soit « la propriété n'est pas vérifiée mais voici un contre-exemple qui aboutit à sa violation ». Cette information constitue un diagnostic utile, directement utilisable par un ingénieur sur la base de sa seule connaissance du système.

Hélas, le *model checking* souffre aussi de quelques inconvénients. Le premier est l'explosion combinatoire du nombre d'états dans des systèmes complexes [VAL 98]. Cela est particulièrement vrai lorsque l'on introduit du parallélisme ou que l'on souhaite analyser des contraintes basées sur le temps. Pour échapper à ce problème, il faut développer des techniques spécifiques qui ne fonctionnent que dans certains cas. L'ingénieur doit donc adapter sa spécification ou utiliser certains outils plutôt que d'autres. Ainsi, le côté automatique et utilisable sans connaissance réelle des techniques sous-jacentes doit être tempéré.

L'autre inconvénient est que le système n'est pas vérifié de manière paramétrée, comme dans le cas d'une preuve, mais uniquement pour une configuration initiale donnée. Cela peut rendre difficile, lorsque c'est nécessaire, l'identification des conditions qui font qu'un système respecte les propriétés voulues.

Par ailleurs, même si le contre-exemple est petit par rapport à la complexité du système, l'utilisateur peut récupérer une trace de 10^8 étapes difficile à analyser, alors que c'est un bon facteur de réduction par rapport à un système comprenant par exemple 10^{80} états.

Dans la suite de cette section, nous nous intéressons aux conditions qui permettent la mise en œuvre d'une vérification par *model checking*.

2.5.3. *Langages d'expression des spécifications*

L'expression des spécifications est un problème crucial car c'est sur cette base que fonctionnent les algorithmes de vérification. On distingue en général deux parties dans la spécification d'un système : le système lui-même et les propriétés qu'il doit respecter. Notons qu'en général, l'expression des propriétés s'avère moins aisée qu'il n'y paraît.

Modélisation du système. Pour être utile à la vérification, elle nécessite l'emploi d'un langage dont la sémantique opérationnelle est formellement définie. Ainsi, le langage doit non seulement être exécutable mais la notion de progression doit être formalisée. Ces langages requièrent en général deux éléments importants : la notion d'état et la notion de transition entre deux états. En particulier, pour appliquer certains algorithmes, il faut qu'une transition entre deux états soit réversible (ce qui n'est pas toujours le cas dans le « monde réel »).

Ainsi, le *model checking* peut difficilement s'appliquer directement sur des langages de programmation dont la sémantique est trop sophistiquée (notons qu'il existe des travaux visant à vérifier directement des programmes, ce qui est évoqué plus loin). Par exemple, l'allocation de mémoire est typiquement un problème complexe à traiter par *model checking*. *A priori*, les automates constituent le langage de base du *model checking*, puisque l'espace d'états n'est autre qu'un automate dont les nœuds sont les configurations (états du système – le vecteur évoqué plus haut) et les transitions les relations entre ces états.

Ainsi, on obtiendra l'espace d'états d'un système au moyen d'un produit entre les automates représentant ses composantes. Mais, comme les automates ne sont pas forcément le modèle le plus facile à utiliser, il est souvent nécessaire de recourir à des « générateurs d'automates » comme les réseaux de Petri [DIA 09], ou des langages aux possibilités restreintes à un modèle comportemental bien défini comme PROMELA [HOL 97, HOL 04], CSP [HOA 85] ou FIACRE [BER 08].

Enfin, des études récentes visent la transformation de langages de haut niveau vers des langages formels. Citons :

- la transformation de programmes C [ZAK 08, JIA 09] ou Java [VIS 05] (en général un sous-ensemble du langage) en programmes PROMELA ;
- la transformation de certains diagrammes UML en réseaux de Petri [KOR 10] ;
- la transformation de spécifications AADL en réseaux de Petri [REN 09] ou en FIACRE [COR 10].

Expression des propriétés. Une fois que nous savons exprimer l'état d'un système, les propriétés peuvent être spécifiées sous la forme d'une formule logique exprimant des contraintes sur les composantes du vecteur décrivant un état. On obtient ainsi une expression atomique permettant de caractériser un profil d'état comme dans la formule ci-dessous qui implique trois variables V_1 , V_2 et V_3 :

$$V_1 < 4 \wedge (V_2 > 5 \vee V_3 = 5)$$

Cette formule peut être évaluée au fur et à mesure de l'exploration de l'espace d'états du système. On parle alors de propriété de sûreté ou d'accessibilité puisqu'il s'agit d'identifier la présence d'un état respectant le profil donné. Lorsque un tel état est atteint, l'algorithme arrête la construction et recherche un chemin entre l'état initial et l'état caractérisé : c'est le contre-exemple. Si l'espace d'états est entièrement parcouru sans que le profil ne soit rencontré, alors le profil n'est pas atteignable. On peut ainsi vérifier l'absence d'états redoutés dans le système.

Mais les expressions atomiques ne permettent pas d'exprimer des propriétés mettant en relation différents profils d'états entre eux. Par exemple :

$$M_{réceptionné} = \text{requête} \Rightarrow \text{dans le futur}, M_{envoyé} = \text{réponse}$$

Cette formule relie tous les états correspondant à la réception d'une requête et les lie au fait que dans le futur, le serveur renvoie forcément une réponse. On parle ici de formules temporelles (au sens causal du terme). De telles formules relient des expressions atomiques au moyen d'opérateurs de logique temporelle [WIK 12]. Il existe plusieurs classes de logiques temporelles, les plus connues étant LTL (*Linear Temporal Logic*) et CTL (*Computation Tree Logic*).

Pour la gestion de propriétés temporisées (c'est-à-dire impliquant du temps), des extensions des logiques temporelles classiques ont été développées comme TCTL ou TLTL. Les opérateurs classiques de CTL (et LTL) peuvent être annotés d'intervalles de temps. Ainsi, la requête précédente, temporisée, devient par exemple :

$$M_{\text{réception}} = \text{requête} \Rightarrow \text{dans moins de 10 unité de temps}, M_{\text{envoyé}} = \text{réponse}$$

La formule ne sera vérifiée que si l'émission de la réponse succède à la réception de la requête dans un intervalle inférieur à dix unités de temps.

Une logique standardisée par l'ISO a été élaboré dans les années 2000 : PSL (*Properties Specification Language*) [EIS 06, IEE 10]. Elle intègre à la fois des notions issues des logiques temporelles classiques et permet de gérer le temps ou des probabilités.

Bien sûr, l'algorithme requise pour évaluer les différents types de formule varie par sa complexité. L'accessibilité est la plus simple (complexité dans la taille de l'espace d'états), viennent ensuite les algorithmes pour les formules de logique temporelle et enfin, celles qui concernent les logiques temporelles temporisées ou probabilistes. Dans certains cas (par exemple, pour les systèmes temporisés), on se heurte à l'indécidabilité de certaines questions : il n'existe pas d'algorithme pouvant systématiquement résoudre le problème.

La principale difficulté de mise en œuvre de l'approche par *model checking* est la capacité des ingénieurs à exprimer les exigences à vérifier sur un système de manière aisée. A ce titre, les logiques temporelles ont une grande expressivité et permettent une expression rigoureuse. Mais en pratique, elles sont difficiles à manipuler dans un contexte industriel car elles demandent une très grande expertise de la part des ingénieurs. En effet, une exigence peut référencer de nombreux évènements, liés à l'exécution du modèle ou de l'environnement, et est dépendante d'un historique d'exécution à prendre en compte au moment de sa vérification.

Une solution à ce problème passe par l'usage de langages dédiés, permettant d'en-cadrer l'expression des propriétés et d'abstraire certains détails, au prix de la réduction de l'expressivité. De nombreux auteurs ont fait ce constat et certains [DWY 99, SMI 02, KON 05] ont proposé de formuler les propriétés à l'aide de patrons de définition. Un patron est une structure syntaxique textuelle qui permet un mode d'expression plus proche des langages que les ingénieurs ont l'habitude de manipuler.

Une autre voie de simplification de l'expression des exigences provient du fait que, dans les documents d'exigences, celles-ci sont souvent exprimées dans un contexte donné de l'exécution du système. Les exigences sont associées à des phases spécifiques d'utilisation du système. Dans [KON 05], les auteurs ont proposé d'identifier la portée (*scope*) d'une propriété en permettant à l'utilisateur de préciser le contexte temporel d'exécution de la propriété à l'aide d'opérateurs (*Global, Before, After, Between, After-Until*). Ces derniers permettent de localiser les exigences à vérifier dans un contexte temporel particulier d'exécution du modèle à valider. Le *scope* indique si la propriété doit être prise en compte, par exemple, durant toute l'exécution du modèle, avant, après ou entre des occurrences d'évènements. L'analyse présentée dans le chapitre 9 (troisième partie) s'inspire de cette notion.

Notons l'existence d'une variante adaptée à la vérification de formules de logique temporelle impliquant des recherches d'événements : l'approche basée sur des observateurs [HAL 93]. L'idée est de surcharger le modèle d'éléments qui ne sont que des observateurs (non intrusifs) d'états particuliers du système. Il est démontré qu'avec de tels observateurs, on peut exprimer ces formules sous la forme d'une propriété d'accessibilité [KUP 99], donc vérifiable par des algorithmes plus simples. Cela nécessite cependant une modification parfois délicate de la spécification. Il faut également s'assurer que cela n'a aucune incidence sur le comportement du système (c'est-à-dire que l'observateur doit rester neutre et ne pas empêcher certains comportements).

Cette démarche a été popularisée avec le langage LUSTRE [HAL 91] et a été reprise dans l'outil UPPAAL [Upp 12]. Si l'expression d'une propriété d'invariance est effectivement plus simple sous la forme d'un automate observateur, l'écriture d'un tel automate peut se révéler malgré tout complexe dans le cas d'une propriété elle-même complexe (un automate observateur réaliste peut rapidement dépasser la vingtaine d'états).

2.5.4. Limites actuelles des approches formelles

Les approches formelles s'imposent progressivement dans bien des secteurs industriels comme une technique de plus en plus incontournable pour assurer une meilleure fiabilité des systèmes. Un indicateur dans ce sens est l'adoption, dans la norme aéronautique DO178C, de techniques formelles pour la conception de systèmes. Cependant, plusieurs questions restent ouvertes à ce jour.

La première concerne la liaison entre le système et le modèle. La vérification s'effectue sur une spécification et non sur le système en lui-même, souvent exprimé dans des termes inappropriés ou de taille trop importante. Il faut donc s'assurer de la cohérence entre les deux, sinon, les propriétés démontrées sur le modèle peuvent ne pas être vraies sur le système qu'il représente. Cela implique la mise en place d'une méthodologie rigoureuse.

Pour assurer cette cohérence entre le système et le modèle, l'ingénierie des modèles propose des approches par transformations (plusieurs sont décrites dans cet ouvrage). La spécification formelle est alors générée à partir d'un modèle de haut niveau qui sert également à la production du code de l'application finale. Cela pose deux problèmes :

- les approches par transformation doivent préserver la sémantique d'exécution entre le modèle source et le modèle cible, ce qui est difficile à démontrer ;
- la taille des spécifications formelles produites devient rapidement très importante : on retombe sur le problème, encore ouvert, de l'explosion combinatoire propre aux approches de vérification par *model checking* (ou leur équivalent pour les approches par preuves).

Enfin, l'usage de méthodes formelles exigeant des outils logiciels complexes, pose le problème de leur certification lorsqu'ils entrent dans la chaîne de production de logiciels eux-mêmes certifiés. Une expérience réussie a été réalisée avec le générateur de code de SCADE qui est certifié et produit donc du code ne nécessitant plus de certification. Cependant, le coût de la certification du générateur de code pour SCADE a un impact négatif sur son utilisation (prix des licences très élevé). Des réflexions sont en cours pour proposer des méthodologies permettant l'usage d'outils non certifiés dans le cadre du développement de logiciels certifiés, les résultats de ces outils ne pouvant probablement plus échapper à une analyse de leur pertinence. Par exemple, pour un générateur de code, cela reviendrait à maintenir une procédure, sans doute simplifiée, de certification du code produit.

Ces éléments, ainsi que le besoin de personnel hautement qualifié, constituent un frein à une adoption plus massive des méthodes formelles. Il n'empêche que ces dernières, tirées par les résultats de recherche, d'une part, par les besoins grandissant en fiabilité des systèmes d'autre part, progressent inexorablement dans le paysage informatique.

2.6. Aspects méthodologiques du développement de systèmes embarqués

Depuis la seconde guerre mondiale, puis avec les grands programmes spatiaux des années 1960, la volonté de mieux maîtriser le développement de systèmes s'est considérablement développée. Pour cela, il faut faire collaborer plus efficacement les équipes et les multiples métiers impliqués dans le processus de fabrication de ces systèmes complexes. Ainsi, les acteurs concernés ont dû formaliser la nature des activités à conduire pour passer d'un besoin esquisssé à un système réel.

L'un des résultats les plus connus est le cycle de développement en V (figure 2.2). C'est devenu un standard de l'industrie logicielle depuis les années 1980. Il distingue :

- une branche descendante de spécification/conception, regroupant toutes les activités allant du besoin à la « définition sur papier » de la solution technique ;

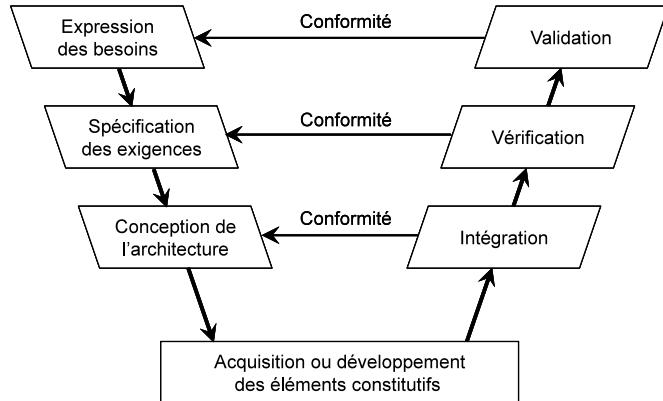


Figure 2.2 – Cycle de développement en V tel que proposé par l’AFIS

- une branche montante couvrant l’intégration, la vérification et la validation du système complet ;
- entre les deux, un palier horizontal correspondant à l’acquisition ou au développement des éléments constitutifs (avec, si nécessaire pour certains, de nouveaux cycles en V imbriqués, la démarche étant itérative).

Cette vision reste très simplificatrice, car elle présente une succession parfaitement séquentielle des activités. Dans la réalité, les retours en arrière et les itérations sont inévitables. Ils peuvent provenir de lacunes ou d’évolutions dans l’expression des besoins (certains aspects ne seront mis en évidence qu’au travers de questions soulevées lors de la conception) comme d’interactions entre les métiers et/ou la recherche des meilleurs compromis, sans parler des problèmes détectés lors de l’intégration ou des essais.

Cela a amené des modèles de cycle de développement plus élaborés (incrémental, en spirale, évolutif, de lignes de produit, etc.), en particulier pour les systèmes dits «à logiciel prépondérant». Aujourd’hui, toutes les normes et standards en ingénierie opère la distinction suivante entre deux dimensions :

- les *phases*, qui jalonnent le projet, en s’appuyant sur un modèle de cycle de développement adapté, et permettent d’en maîtriser le déroulement (par exemple, la phase de spécification du système, qui se solde par la fourniture de la spécification technique)
- les *processus* (ensemble d’activités liées), qui indiquent les activités à conduire ; ces dernières sont, pour l’essentiel, indépendantes du type de système et du domaine d’activité considérés.

Un même type d'activité peut concerner plusieurs phases. Ainsi, des activités de conception peuvent-elles être nécessaires très en amont, par exemple pour disposer de maquettes d'illustration et d'analyse des besoins. De même, les processus de vérification et de validation s'appliquent principalement au système final, mais peuvent aussi concerner les résultats d'autres phases afin de détecter des problèmes et des dérives au plus tôt (typiquement l'oubli de besoins/contraintes ou l'impact d'un événement redouté). L'objectif est de limiter l'impact de ces problèmes sur les coûts et les délais de production du système. La gestion de projet doit sélectionner, organiser et planifier les activités à conduire durant chaque phase.

2.6.1. *Les principaux processus techniques*

Les principaux processus techniques considérés dans les normes et standards d'ingénierie sont, à des variantes de périmètre et de vocabulaire près, basés sur le même schéma. Notons que ces processus indiquent ce qu'il faut faire, mais pas comment le faire. C'est l'objet des méthodes, qui sont très dépendantes des métiers et des domaines d'application. Nous trouvons donc dans les processus actuels, les éléments suivants :

- l'expression des besoins : il s'agit de recueillir, analyser et formaliser avec l'ensemble des parties concernées (clients, utilisateurs et développeurs) les besoins que devra respecter le système futur ;
- la spécification des exigences : il s'agit de traduire les besoins en termes techniques mesurables, indépendamment de toute idée de solution. Le système est alors vu comme une « boîte noire » dont les interactions avec son environnement sont finement analysées. C'est à cette étape que sont également précisées les exigences « non fonctionnelles » (sûreté de fonctionnement, sécurité, fiabilité, disponibilité, etc.) ainsi que d'éventuelles contraintes spécifiques (réglementaires, environnementales, etc.) ;
- la conception fonctionnelle : cette étape couvre la définition de l'architecture fonctionnelle interne de la solution (indépendamment de toute contrainte technologique), l'allocation des exigences aux fonctions ainsi que la description des caractéristiques et du comportement attendus des fonctions ;
- la conception physique : cette étape couvre le partitionnement des fonctions (on parle de l'architecture logique), la définition de l'architecture physique de la solution et de l'allocation de fonctions à des éléments (matériels, logiciels et, éventuellement, humains). On y spécifie les composants à développer ainsi que leurs interfaces ;
- l'intégration du système : il s'agit de rassembler en un système unique l'ensemble des constituants développés séparément jusqu'à ce stade ;
- la vérification² du système : Cette étape concerne les actions de vérification incomptant au concepteur/réalisateur du système. Elles lui permettent de s'assurer qu'il

2. Ne pas confondre avec la « vérification formelle » au sens de la section 2.5.

a « bien fait » le produit, c'est-à-dire : respect du référentiel des exigences, absence de défauts vis-à-vis des règles de l'art et des normes, etc ;

– la validation du système : il s'agit des actions permettant au client et à l'utilisateur final de s'assurer que le produit est « bon », c'est-à-dire qu'il répond correctement au référentiel des besoins identifiés lors de la première étape (expression des besoins).

2.6.2. L'importance des modèles

Le développement de systèmes nécessite la collaboration de multiples acteurs, d'équipes parfois importantes et de métiers très variés. Il faut donc des supports afin d'assurer la compréhension mutuelle de tous. C'est le rôle des modèles qui sont des représentations abstraites et partielles du système selon un angle de vue et avec un niveau de granularité permettant d'en étudier certaines caractéristiques (propriétés, comportement, etc.). Les modèles sont donc élaborés avec un objectif précis ; aucun d'eux ne suffit à lui seul à traduire la complexité d'un système.

De nombreux langages de modélisation, plus ou moins spécialisés, existent. Ce livre s'intéresse aux langages adaptés à la description et à l'analyse des systèmes embarqués, lesquels se caractérisent par des exigences élevées en matière de temps de réponse, de sûreté et de sécurité :

– SysML est un langage de modélisation graphique pour capturer les aspects structuraux, fonctionnels et comportementaux des systèmes incorporant du matériel, du logiciel, des hommes, des procédures. Il est utilisable en support des activités de spécification, d'analyse, de conception et de vérification-validation ;

– UML/MARTE et AADL sont des langages de modélisation et d'analyse dédiés aux systèmes embarqués temps réel. Ils permettent de prendre en compte les propriétés non fonctionnelles qui les caractérisent (contraintes de temps, de sûreté, de sécurité, etc.) et de vérifier des propriétés telles que l'ordonnancement, la bonne transmission des messages, le bon dimensionnement du matériel. Ils se positionnent plutôt en fin des activités de conception. Ce ne sont ni des méthodologies, ni des outils.

L'ingénierie dirigée par les modèles a pour ambition, en donnant un rôle central aux modèles, de garantir la cohérence des éléments manipulés tout au long du cycle de développement par les diverses parties prenantes. L'idée est d'affiner progressivement les modèles tout au long de l'analyse des besoins, de la spécification et de la conception et de s'appuyer sur des techniques de transformation de modèles pour garantir le maintien de leur cohérence et la préservation des propriétés à chaque étape.

Les différences d'ordre sémantique entre les langages de modélisation propres à certains domaines et l'interopérabilité limitée entre outils constituent encore des freins à cette ambition.

2.7. Conclusion

Dans ce chapitre, nous avons dressé un panorama de la modélisation et de l'analyse de systèmes embarqués. Loin d'être exhaustif, ce panorama vise avant tout à montrer la richesse de ce contexte : variété des méthodes de description, des techniques d'analyse et la nécessité de les combiner *via* un processus rationalisé en lien avec des contraintes (métiers, normatives et réglementaires).

Ce chapitre permettra au lecteur de mieux situer chacune des notations présentées. Le livre est articulé autour d'un même plan : présentation de la notation, utilisation pour représenter un problème complexe : un pacemaker, analyse et génération de code. La lecture de ces chapitres permettra ainsi de situer ces différentes notations.

Au premier abord, on peut considérer que SysML est utile dans les phases amont de modélisation. SysML étant avant tout une notation pour l'ingénierie système, il faudra le compléter par une notation plus proche des considérations d'implantation. Il faut dès lors choisir, UML/MARTE ou AADL :

- UML/MARTE est un candidat naturel du fait de la filiation UML qu'il partage avec SysML. UML/MARTE permettra de poursuivre la modélisation du système, en rendant plus explicite certains aspects du système. Néanmoins, il souffre de nombreuses limitations liées à UML : manque de processus clair, libertés d'interprétation et une sémantique plus large qu'il faudra dans certains cas restreindre ;
- AADL permet quant à lui une modélisation plus précise, avec une sémantique restreinte aux systèmes embarqués critiques. De plus, il dispose d'ores et déjà de nombreux outils d'analyse. Néanmoins, comme il est hors cadre UML, la traçabilité entre SysML et AADL doit être construite.

On peut imaginer ainsi plusieurs couplages : UML/MARTE permettant de modéliser PIM puis PSM du système considéré, puis AADL afin d'avoir une vision consolidée de tous les éléments, préparant le travail d'intégration, de vérification et validation. Ou encore se limiter à une seule notation, UML/MARTE ou AADL pour la modélisation, l'analyse et la génération du système raffiné.

On l'aura compris, le choix d'une notation n'est pas simple, et est avant tout un choix qui doit être dicté par le problème à résoudre, les outils mis à disposition et la familiarité de l'ingénieur avec la notation.

C'est la raison d'être de ce livre : vous permettre de faire un tel choix en traitant d'un même cas d'étude au moyen de ces trois notations.

2.8. Bibliographie

- [ABR 96] ABRIAL J.-R., *The B book - Assigning Programs to meanings*, Cambridge Univ. Press, 1996.

- [AFF 08] AFFELDT R., KOBAYASHI N., « A Coq Library for Verification of Concurrent Programs », *Electronic Notes in Theoretical Computer Science (ENTCS)*, vol. 199, p. 17-32, 2008.
- [BER 08] BERTHOMIEU B., BODEVEIX J.-P., CHAUDET C., DAL ZILIO S., FILALI M., VERNADAT F., « Verifying Dynamic Properties of Industrial Critical Systems Using TOPCASED/FIACRE », *ERCIM NEWS, Special Issue on Safety-Critical Software*, Septembre 2008.
- [CLA 86] CLARKE E., EMERSON E., SISTLA A., « Automatic verification of finite-state concurrent systems using temporal logic specifications », *ACM Trans. Program. Lang. Syst.*, vol. 8, n° 2, p. 244–263, ACM, 1986.
- [CLA 00] CLARKE E., GRUMBERG O., PELED D., *Model Checking*, MIT Press, 2000.
- [COR 10] CORREA T., BECKER L. B., FARINES J.-M., BODEVEIX J.-P., FILALI M., VERNADAT F., « Supporting the Design of Safety Critical Systems Using AADL », *15th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS)*, IEEE Computer Society, p. 331-336, 2010.
- [DEM 11] DEMRI S., POITRENAUD D., « Verification of Infinite-State Systems », HADDAD S., KORDON F., PAUTET L., PETRUCCI L., Eds., *Models and Analysis in Distributed Systems*, Chapitre 8, p. 221-269, Wiley, 2011.
- [DIA 09] DIAZ M., Ed., *Petri Nets, Fundamental Models, Verification and Applications*, Wiley-ISTE, 2009.
- [DWY 99] DWYER M. B., AVRUNIN G. S., CORBETT J. C., « Patterns in property specifications for finite-state verification », *21st Int. Conf. on Software Engineering*, IEEE Computer Society Press, p. 411-420, 1999.
- [EIS 06] EISNER C., FISMAN D., *A Practical Introduction to PSL*, Series on Integrated Circuits and Systems, Springer, 2006.
- [FIM 74] FIMMEL R. O., SWINDELL W., BURGESS E., « SP-349/396 PIONEER ODYSSEY », available at history.nasa.gov/SP-349/contents.htm, 1974.
- [HAL 91] HALBWACHS N., CASPI P., RAYMOND P., PILAUD D., « The synchronous dataflow programming language LUSTRE », *Proceedings of the IEEE*, p. 1305–1320, 1991.
- [HAL 93] HALBWACHS N., LAGNIER F., RAYMOND P., « Synchronous observers and the verification of reactive systems », NIVAT M., RATTRAY C., RUS T., SCOLLO G., Eds., *Third Int. Conf. on Algebraic Methodology and Software Technology, AMAST'93*, Twente, Workshops in Computing, Springer Verlag, June 1993.
- [HOA 85] HOARE C., *Communicating Sequential Processes*, Prentice-Hall, 1985.
- [HOL 97] HOLZMANN G., « The Model Checker SPIN », *Software Engineering*, vol. 23, n° 5, p. 279-295, 1997.
- [HOL 04] HOLZMANN G., « The SPIN model checker », Chapitre An Overview of PROMELA, p. 33-72, Addison-Wesley, 2004.
- [IEEE 10] IEEE 1850, « IEEE Standard for Property Specification Language (PSL) », 2010.

- [ISO 02] ISO/IEC 13568, « Z formal specification notation — Syntax, type system and semantics », 2002.
- [JIA 09] JIANG K., JONSSON B., « Using SPIN to Model Check Concurrent Algorithms, using a translation from C to Promela », *2nd Swedish Workshop on Multi-Core Computing*, 2009.
- [KON 05] KONRAD S., CHENG B., « Real-Time Specification Patterns », *27th Int. Conf. on Software Engineering (ICSE05)*, St Louis, MO, USA, 2005.
- [KOR 10] KORDON F., THIERRY-MIEG Y., « Experiences in Model Driven Verification of Behavior with UML », *Foundations of Computer Software, Future Trends and techniques for Development, 15th Monterey Workshop 2008, Budapest, Revised Selected Papers*, vol. 6028 de *Lecture Notes in Computer Science*, Springer, p. 181-200, 2010.
- [KUP 99] KUPFERMAN O., VARDI M., « Model Checking of Safety Properties », *International Conference on Computer Aided Verification*, vol. 1633 de *Lecture Notes in Computer Science*, p. 685-685, Springer, 1999.
- [OMG 12a] OMG, « The Official OMG SysML site », www.omg.org 2012.
- [OMG 12b] OMG, « UML Profile For MARTE : Modeling And Analysis Of Real-Time Embedded Systems », www.omg.org/spec/MARTE 2012.
- [QUE 82] QUEILLE J.-P., SIFAKIS J., « Specification and verification of concurrent systems in CESAR », *Proceedings of the 5th Colloquium on International Symposium on Programming*, London, UK, Springer-Verlag, p. 337–351, 1982.
- [REN 09] RENAULT X., KORDON F., HUGUES J., « Adapting models to model checkers, a case study : Analysing AADL using Time or Colored Petri Nets », *Proceedings of the 20th International Symposium on Rapid System Prototyping*, Paris, IEEE Computer Society, p. 26-33, June 2009.
- [SAE 09] SAE, « Architecture Analysis & Design Language V2 (AS5506A) », available at www.sae.org 2009.
- [SMI 02] SMITH R., AVRUNIN G., CLARKE L., OSTERWEIL L., « Propel : An Approach Supporting Property Elucidation », *24st Int. Conf. on Software Engineering (ICSE02)*, St Louis, MO, USA, ACM Press, p. 11-21, 2002.
- [Upp 12] UPPSALA & AALBORG UNIVERSITIES, « UPPAAL Home », www.uppaal.org 2012.
- [VAL 98] VALMARI A., « The State Explosion Problem », REISIG W., ROZENBERG G., Eds., *Lectures on Petri Nets 1 : Basic Models*, vol. 1491 de *Lecture Notes in Computer Science*, p. 429–528, Springer-Verlag, 1998.
- [VIS 05] VISSER W., MEHLITZ P. C., « Model Checking Programs with Java PathFinder », *Model Checking Software, 12th International SPIN Workshop*, vol. 3639 de *Lecture Notes in Computer Science*, Springer, page 27, 2005.
- [WIK 12] WIKIPEDIA, « Temporal Logic », en.wikipedia.org/wiki/Temporal_logic 2012.
- [ZAK 08] ZAKS A., JOSHI R., « Verifying Multi-threaded C Programs with SPIN », *15th International SPIN Workshop*, vol. 5156 de *Lecture Notes in Computer Science*, Springer, p. 325-342, 2008.

Chapitre 3

Etude de cas : le pacemaker

3.1. Introduction

L'objectif de ce chapitre est de présenter l'étude de cas qui servira de fil rouge aux différentes techniques de modélisation présentées dans cet ouvrage. Nous avons retenu comme cas d'étude un pacemaker dont les spécifications ont été publiées par la *société Boston Scientific* [Bos 07]. Ce cas d'étude fait partie d'un grand challenge dont le but est de stimuler la recherche autour des méthodes formelles et les techniques de modélisation associées, sous la forme d'une série de challenges aux communautés académiques et industrielles [WOO 06].

Ce cas d'étude a été traité sous de multiples formes, on peut citer une modélisation en Z [GOM 09], B-événementiel [MÉR 09], VDM [MAC 08], algèbre de processus [TUA 10]. Finalement, l'équipe vainqueur du concours a publié en 2009 le rapport détaillant leur approche, combinant preuve de théorèmes (dont PVS [MAN 09]) et solveur SAT, disponible sur le site de SCORE¹. Parallèlement, une carte a été développée par l'Université du Minnesota [NIX 09].

Nous avons retenu ce cas d'étude pour cet ouvrage car il fournit un exemple concret d'un système embarqué connu, couvrant un large spectre de préoccupations allant des aspects de procédure (installation, mise en route du pacemaker), des interactions entre opérateur/patient/dispositif, et son fonctionnement.

Chapitre rédigé par Fabrice KORDON, Jérôme HUGUES, Agusti CANALS et Alain DOHET.

1. score-contest.org/projects.php#Pacemaker

3.2. Le cœur, le pacemaker

Dans cette section, nous présentons brièvement le système étudié : un stimulateur cardiaque ou pacemaker. Un pacemaker est un petit système électronique dont le but est d'aider le cœur à maintenir un rythme régulier. Le pacemaker est implanté dans le torse du patient lors d'un acte chirurgical. Des sondes sont placées sur le muscle cardiaque. Le pacemaker lui-même est placé sous la peau, proche de l'épaule. Le pacemaker opère dans l'environnement immédiat du cœur.

Nous présentons d'abord le fonctionnement du cœur avant de revenir sur les éléments d'un pacemaker.

3.2.1. *Le cœur*

Le rôle du cœur est de permettre au sang d'irriguer les différents organes du corps humain. Il agit comme un pompe fonctionnant sans discontinuer tout au long de la vie. Ainsi, on peut ramener le fonctionnement du cœur à celui d'un système mécanique et électrique. Son comportement est régulé en fonction de l'effort. Ainsi, un « algorithme » de contrôle existe.

Au repos (c'est-à-dire en absence d'effort physique), la fréquence cardiaque est de soixante à huit battements par minute pour un débit de 4,5 à 5 litres de sang par minute. Au cours d'une vie, le cœur peut battre plus de deux milliards de fois.

Le système mécanique du cœur (la pompe) est stimulé électriquement pour déclencher le mouvement des muscles. Le cœur (voir la figure 3.1) est formé de quatre compartiments : les ventricules et atria (ou oreillettes) gauche et droit. Ces compartiments se contractent et se rétractent périodiquement sous le contrôle de stimuli électriques. Les atria forment une unité, les ventricules une seconde unité.

Chacun de ses mouvements entraîne une séquence d'événements collectivement appelés la « révolution cardiaque ». Celle-ci consiste en trois étapes majeures : la systole auriculaire, la systole ventriculaire et la diastole :

- au cours de la systole auriculaire, les oreillettes se contractent et éjectent du sang vers les ventricules (remplissage actif). Une fois le sang expulsé des oreillettes, les valves auriculo-ventriculaires entre les oreillettes et les ventricules se ferment. Le sang continue tout de même à affluer dans les oreillettes. Ceci évite un reflux du sang vers les oreillettes. C'est la fermeture de ces valves qui produit le son familier du « battement » du cœur ;
- la systole ventriculaire implique la contraction des ventricules, expulsant le sang vers le système circulatoire. En fait, dans un premier temps, très bref, les valvules sigmoïdes sont fermées. Dès que la pression à l'intérieur des ventricules dépassent la pression artérielle, les valvules sigmoïdes s'ouvrent. Une fois le sang expulsé, les

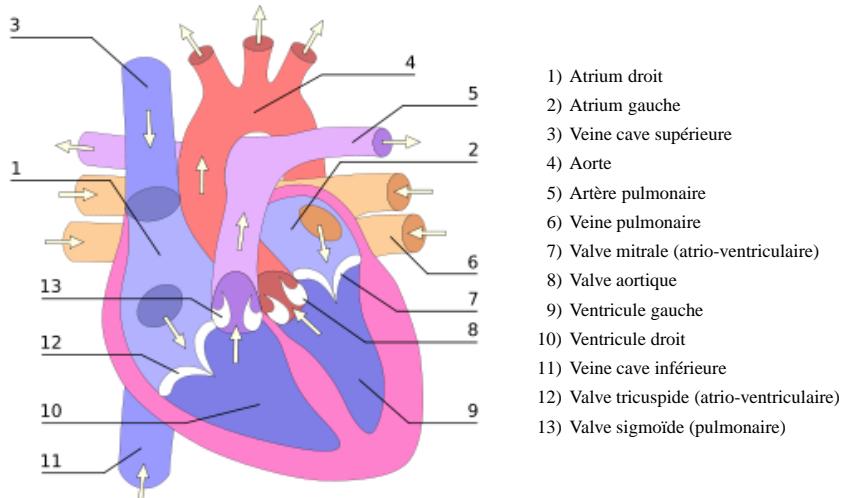


Figure 3.1 – Planche anatomique du cœur, ©Wikipedia

deux valves sigmoïdes – la valve pulmonaire à droite et la valve aortique à gauche – se ferment. Ainsi le sang ne reflue pas vers les ventricules. La fermeture des valvules sigmoïdes produit un deuxième bruit cardiaque plus aigu que le premier. La pression sanguine augmente ;

– la diastole est la relaxation de toutes les parties du cœur, permettant le remplissage (passif) des ventricules (plus de 80 % du remplissage dans les conditions usuelles), par les oreillettes droite et gauche et depuis les veines cave et pulmonaire. Les oreillettes se remplissent doucement et le sang s'écoule dans les ventricules.

En fonctionnement normal, le corps humain a en charge la stimulation du cœur *via* des mécanismes de contrôle par le système nerveux central et le système hormonal. Ces deux systèmes jouent le rôle d'un pacemaker « naturel ». Deux nœuds du système nerveux jouent un rôle-clé dans la stimulation électrique : le nœud sinusal et le nœud atrio-ventriculaire. Le nœud sinusal reçoit une première décharge électrique, de l'ordre du millivolt qui est ensuite transmise au nœud atrio-ventriculaire qui l'amplifie, stimulant les ventricules.

Un pacemaker peut être implanté à un patient dans le cas où le corps humain ne peut assurer seul un fonctionnement régulier du cœur, cohérent avec l'effort physique. Dans ce cas, le pacemaker vient se substituer aux ordres habituellement transmis par le corps. Nous présentons son fonctionnement dans la section suivante.

3.2.2. Présentation d'un pacemaker

Un pacemaker est un dispositif électronique implanté dans le cœur d'un patient souffrant de bradycardie (du grec *bradus* = lent et *kardia* = cœur). La bradycardie est une pathologie qui se caractérise par un rythme cardiaque trop bas par rapport à la normale, en général inférieur à soixante pulsations par seconde, ou à l'inverse de tachycardie, c'est-à-dire d'un rythme cardiaque trop élevé, supérieur à une centaine de pulsations en l'absence d'effort physique significative.

Le but d'un pacemaker est d'émettre des signaux électriques directement sur le cœur. Le cœur étant un muscle, la réception de ces signaux correctement dimensionnés vont corriger le mouvement.

Les éléments de base d'un pacemaker sont :

- sondes : un ou plusieurs fils métalliques enroulés, le plus souvent deux qui transmettent un signal électrique entre le cœur et le pacemaker. Chaque sonde peut avoir un ou deux points de contact avec le cœur, on parle alors de sonde uni ou bipolaire ;
- générateur de pulsation : le pacemaker est à la fois la source de puissance, mais aussi le contrôleur des battements du cœur et son moniteur. Il contient donc un accumulateur et une unité générant un signal et analysant les battements du cœur, et éventuellement émettant un signal électrique. Il est implanté au plus près du cœur ;
- moniteur-contrôleur externe² : il s'agit d'une unité externe qui interagit avec le pacemaker *via* une connexion sans fil. Il contient la partie applicative du pacemaker, qui prend la décision d'émettre ou non un signal en fonction des battements du cœur du patient ;
- accéléromètre : placé dans le pacemaker, il mesure l'accélération et les mouvements du corps du porteur afin d'ajuster les stimulations.

Dans le cas d'un pacemaker à électrode simple, celle-ci est attachée à l'atrium pour le ventricule droit. Le pacemaker dispose de plusieurs modes opérationnels pour contrôler le mouvement du cœur, rappelées dans le document de spécification [Bos 07]. Ces paramètres sont liés à des contraintes temps réel et de causalité (action/réaction) permettant de contrôler le rythme cardiaque.

Cette présentation rapide du pacemaker ne couvre pas ces nombreux cas d'utilisation, et les multiples interactions qui peuvent survenir entre patient, pacemaker et équipe médicale. En particulier, nous n'avons pas présenté les phases implantatoires, diagnostic et cas d'erreurs associés. Nous recommandons l'ouvrage de Barold *et al.* [BAR 10] pour une présentation plus détaillée.

2. Référencé comme le *Device Controller-Monitor (DCM)* dans les spécifications originales.

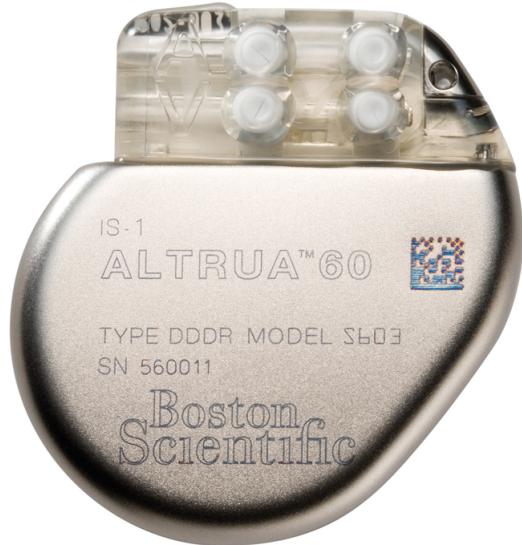


Figure 3.2 – Pacemaker de *Boston Scientific*

3.3. Spécification du cas d'étude

Dans cette section, nous nous intéressons avant tout à la description du pacemaker du document de spécification [Bos 07], et ne traiterons donc pas de la partie interface graphique présentée en section 4 de ce document.

Cette section vise aussi à introduire les exigences du système qui seront ensuite exploitées par les différents langages de modélisation proposés dans cet ouvrage. Les exigences sont numérotées, et indiquées comme suit :

(1)/S (*But du client*) Le pacemaker fonctionne correctement.

Un identifiant dans la marge «(XX)/Y» indique que le paragraphe concerné est une exigence, définie en langage naturel. Un texte en italique en début de paragraphe définit la catégorie du besoin. Ainsi, l'exigence 1 définit une exigence naïve : le pacemaker fonctionne correctement. Le suffixe indique à quel niveau se place l'exigence, «S» pour le système, «P» pour le pacemaker lui-même.

Il nous faut maintenant raffiner cette exigence, et clarifier ce que «correctement» signifie. C'est l'objet des sections suivantes.

Nous reproduisons ici une traduction des exigences telles qu'elles ont été définies par *Boston Scientific*. Le lecteur attentif notera que certaines exigences sont en fait

des définitions, que nous avons conservées. C'est au processus de conception que reviendra la charge de trier ces exigences.

3.3.1. Définition du système

Le pacemaker auquel nous nous intéressons répond aux besoins suivants des patients :

- implantation ;
- phase ambulatoire, lors de la phase d'étalonnage du système ;
- suivi du patient ;
- retrait (explantation) du pacemaker.

Ce système fournit des stimulations dont le rythme s'adapte au patient, sur les deux chambres du cœur ; des données sur son fonctionnement dans un journal de suivi, et un diagnostique de son bon fonctionnement.

Les fonctions d'analyse permettent de mesurer les éléments suivants : impédance des sondes, seuil de pulsation, mesures du cœur, état de la batterie, émission d'une pulsation, étalonnage de l'accéléromètre.

Le pacemaker étant un système médical, ses composants visibles sont définis et standardisés, facilitant l'échange entre patients et équipes médicales.

Ainsi, le pacemaker est composé de trois composants principaux : le pacemaker à proprement parler, c'est-à-dire le générateur de pulsations (PG – *Pulse Generator*) ; le contrôleur-moniteur (DCM – *Device Controller Monitor*) et son logiciel de contrôle associé ; les sondes. L'aimant cardiaque en forme de beignet (un *donut*) est un composant mineur. Notons que par abus de langage, on assimile le plus souvent le pacemaker au seul générateur de pulsations.

Le générateur de pulsations (PG) contrôle et régule les pulsations cardiaques du patient, fournissant ainsi une thérapie pour les bradychardies. Le PG fournit un moyen programmable pour stimuler une ou deux chambres cardiaques, adapté au rythme du patient, que ce soit de manière permanente ou temporaire.

Dans le mode adaptable, un accéléromètre est utilisé pour mesurer l'activité physique du patient. Cette mesure est utilisée pour calculer la stimulation à émettre.

Le PG est programmé et interrogé par télématiétrie par le contrôleur (DCM). Ceci permet au médecin de changer le mode opérationnel ou certains paramètres de manière non intrusive après implantation.

Le PG fournit les données suivants : données des capteurs ; histogramme des *atria* et ventricules.

Le PG, en conjonction avec le DCM fournit les informations de diagnostique suivantes : télémétrie en temps réel ; électrogrammes (EGM) ; mesures du cœur ; impédance des sondes ; état des batteries.

Le contrôleur-moniteur (DCM) est le support maître de la thérapie ; il est utilisé par les différentes équipes médicales (implantation, ambulatoire, suivi). Il communique par un protocole de communication et une couche matérielle dédiée. Le DCM est composé d'une plate-forme matérielle et d'un bloc logiciel programmable.

Le DCM fournit les fonctionnalités suivantes : programmer et interroger le pacemaker ; envoyer un ordre de stimulation immédiates ; gestion de l'historique des valeurs des capteurs ; affichage de l'historique et des informations d'étalonnage.

Par ailleurs, il dispose de fonctions auxiliaires : indicateur de bon fonctionnement *via* des LED, acquisition de données pour un électrocardiogramme ; suivi de l'état de la batterie ; sortie vers un dispositif d'affichage, etc. Par la suite, nous ne considérons pas ces fonctions.

Les sondes implantées sur le cœur du patient permettent de détecter l'activité électrique du cœur, et le stimulent. Les sondes sont connectées au PG *via* une connectique adaptée. Toutes les sondes bipolaires IS-1 sont supportées.

Le pacemaker et les sondes sont implantés par une équipe médicale à l'hôpital. Le suivi des patients est effectué par des infirmiers et techniciens, sous la supervision d'un médecin.

Le pacemaker est conforme aux exigences en matière de sûreté électrique. Il doit fonctionner lors des différentes étapes de son cycle de vie, qui peuvent être sources d'interférences : fluoroscope, machines de contrôle lors de l'anesthésie, poches d'eau, défibrillateurs, etc.

3.3.2. Cycle de vie du système

(2)/S Le cycle de vie du pacemaker est le suivant :

- la phase préimplantatoire : le système est assemblé suivant les bonnes pratiques édictées par les organismes de certification et de régulation sanitaires. Durant cette phase, les paramètres nominaux du PG sont configurés ;
- l'implantation : le pacemaker est placé à l'intérieur du patient. Lors de cette phase, l'unité DCM est utilisée pour interroger le système, vérifier les batteries, tester le pacemaker, le configurer, programmer le système, évaluer les différents paramètres lus sur les sondes. La procédure implantatoire prévoit les étapes suivantes : vérification de tous les équipements, implantation des sondes, évaluation des signaux au niveau des sondes, programmation du système avant implantation, mise en place des sondes dans le corps, connexion des sondes et tests, implantation du PG ;

52 Systèmes embarqués

- le suivi préliminaire : durant cette phase, une série de tests peut être effectuée : interroger le système pour connaître ses paramètres, le reprogrammer, imprimer le rapport d'état du système pour archivage auprès du dossier patient ;
- le suivi de routine : le système programmable peut effectuer les actions suivantes, dites de routine : interroger l'état du système, vérifier l'état de la batterie, des paramètres de bradycardie du patient, mesurer les ondes P et R, tester le seuil de stimulation et d'impédance des sondes, passer en revue l'historique des capteurs et histogrammes associés et les effacer, imprimer le rapport de tests. Si les paramètres du PG sont modifiés, ceux-ci peuvent être vérifiés avant leur implantation ;
- ambulatoire : les fonctions de stimulations et de mesure doivent être disponibles ;
- l'explantation : une fois le PG retiré, il est nettoyé et retourné à son fabricant. Une analyse de panne est éventuellement pratiquée. L'état du PG, lorsqu'il est retiré, dépend des pannes qu'il a pu subir et/ou de l'état de la batterie ;
- la destruction : le pacemaker est renvoyé à son fabricant. Il ne doit pas être incinéré du fait de certains composants pouvant exploser en présence de fortes chaleurs.

3.3.3. Exigences système

Dans la suite, nous considérons le système pacemaker pris seul, dans le cas du traitement de la bradycardie.

- (3)/S Le pacemaker doit être clairement identifié par un numéro de modèle, sa désignation, les fonctionnalités supportées et les connecteurs utilisés.
- (4)/S Le pacemaker dispose d'un manuel de fonctionnement et d'un logiciel de configuration, suivi des mesures. Nous ne présentons pas cette partie.
- (5)/S Le pacemaker le DCM doit utiliser une télémétrie par induction pour communiquer avec le générateur de pulsations maintenant une communication à une distance inférieure à 5 cm ; ou un dispositif ultrasons ou par radiofréquence dans les limites légales.
- (6)/S Des sondes bipolaires placées sur l'atrium et le ventricule doivent être utilisables. L'impédance des sondes doit être comprise entre 100Ω et $2\,500\Omega$.
- (7)/S Les pulsations générées par le pacemaker doivent être programmables en amplitude et en largeur.
- (8)/S La mesure des pulsations doit se faire à l'aide d'électrodes. La décision de détecter un battement doit être basée sur la mesure des longueurs de cycle. La mesure doit s'effectuer sur une fenêtre glissante.
- (9)/S La sensibilité des sondes doit être un paramètre ajustable par le médecin. Chaque sonde doit être programmable indépendamment.

Le pacemaker est un dispositif électronique programmable. Son comportement général est invariant, et est défini par le fonctionnement intrinsèque du cœur. En revanche, certaines situations sont ajustables en fonction de la pathologie du patient. Les modes bradycardes énumèrent les différentes situations pour lesquelles le pacemaker peut être programmé.

- (10)/S (*Nommage*) Le mode opérationnel d'un pacemaker est défini par un code universel composé de trois ou quatre caractères. Ce code fournit une indication claire sur les fonctions activées à un instant donné. Cette séquence est appelée « mode opérationnel du pacemaker ». Chaque chambre est associée à une lettre : « O » : rien, « A » : atrium, « V » : ventricule, « D » : équivaut à « A » + « V ».

Dans ce code, la première lettre représente la chambre stimulée, la seconde celle qui est mesurée, la troisième pour indiquer si l'on doit réagir à une mesure et la dernière lettre, optionnelle, indique qu'il faut moduler le rythme cardiaque du fait d'une activité physique, mesurée par l'accéléromètre. La lettre « X » est utilisée pour indiquer toutes les autres lettres (c'est-à-dire « O », « A », « V » ou « D »). La lettre « T » (*triggered*, déclenché en français) indique la délivrance d'un stimulus, un signal électrique, tandis que « I » (*inhibited*, inhibé en français) indique son absence après avoir mesuré une activité intrinsèque d'une des chambres du cœur.

Les exigences qui suivent définissent le comportement à court terme du pacemaker, c'est-à-dire son comportement sur une période de temps de l'ordre de quelques battements cardiaques.

- (11)/S (*Pas de réponse à une mesure (O)*) Stimulation sans mesure, ou stimulation asynchrone, correspond à une stimulation délivrée sans tenir compte des valeurs mesurées.

- (12)/S (*Réponse déclenchée (T)*) Dans ce mode, le résultat d'une mesure dans une chambre doit déclencher immédiatement une stimulation dans cette même chambre.

- (13)/S (*Réponse inhibée (I)*) Dans ce mode, une mesure sur une chambre doit inhiber une stimulation en attente dans cette même chambre.

- (14)/S (*Réponse suivie (D)*) Dans ce mode, une mesure doit déclencher une stimulation du ventricule après un délai programmé AV, à moins qu'un mouvement de ce même ventricule soit détecté dans l'intervalle.

Le pacemaker définit une série de modes à plus longue échéance, définissant un état courant pour le patient : permanent, temporaire, stimulation immédiate, test aimant et remise à l'état initial (POR, *Power-On Reset*). Ces modes sont mutuellement exclusifs.

- (15)/S (*Etat permanent*) Cet état est le mode de fonctionnement normal du pacemaker. Les paramètres configurés pour le patient doivent être utilisés.

(16)/S (*Etat temporaire de bradycardie*) Cet état correspond à un jeu de paramètres qui permettent de tester les paramètres du pacemaker et de fournir un diagnostic du patient. Ce mode est terminé par l'une des actions suivantes : une stimulation immédiate, une commande de l'unité DCM.

(17)/S (*Stimulation immédiate*) Une stimulation bradycarde d'urgence commandée (*Pace-Now*) doit être disponible. Les paramètres sont les suivants :

- le mode de la stimulation doit être « VVI » ;
- la limite basse de la stimulation doit être de $65 \text{ ppm} \pm 8 \text{ ms}$;
- l'amplitude de la stimulation doit être de $5 \text{ V} \pm 0,5 \text{ V}$;
- la largeur de la stimulation est de $1 \text{ ms} \pm 0,02 \text{ ms}$;
- la période refractaire ventriculaire est de $320 \text{ ms} \pm 8 \text{ ms}$;
- la sensibilité ventriculaire est de $1,5 \text{ mV}$;
- la première stimulation doit être émise au plus tôt après deux cycles plus 500 ms depuis la dernière activation ;
- une fois activé, ce mode doit être maintenu jusqu'à contre-ordre du DCM.

(18)/S (*Test de l'aimant*) C'est l'état propre au test de l'aimant.

(19)/S (*Remise à l'état initial*) Cette opération est effectuée lorsque le niveau de la batterie est trop bas pour maintenir un comportement efficace du système. Toutes les fonctions doivent être désactivées jusqu'à ce que la batterie atteigne à nouveau un seuil suffisant. Une fois ce seuil atteint, les paramètres du pacemaker sont les suivants :

- le mode de la stimulation doit être « VVI » ;
- la limite basse de la stimulation doit être de $65 \text{ ppm} \pm 8 \text{ ms}$;
- l'amplitude de la stimulation doit être de $5 \text{ V} \pm 0,5 \text{ V}$;
- la largeur de la stimulation est de $0,5 \text{ ms} \pm 0,02 \text{ ms}$;
- la période refractaire ventriculaire est de $320 \text{ ms} \pm 8 \text{ ms}$;
- la sensibilité ventriculaire est de $1,5 \text{ mV}$;

(20)/S (*Test Aimant*) Ce mode est utilisé pour déterminer le statut de la batterie. Un aimant standard, ayant la forme d'un *donut*, doit être détecté par le pacemaker à une distance de $2,5 \text{ cm}$ de la surface du pacemaker.

Lorsque l'aimant est en place, le périphérique doit :

- stimuler de manière asynchrone à un rythme fixe. Le mode du pacemaker doit être « AOO » si le mode précédent était « AXXX », « VOO » si le mode précédent était « VXXX », « DOO » si le mode précédent était « DXXX », ou « OOO » si le mode précédent était « OXO » ;

- au démarrage du pacemaker, le rythme doit être de cent battements par seconde ; lorsque le niveau de la batterie a baissé jusqu'à un certain seuil, le rythme descend à 90 ppm, puis 85 ppm, et continue de baisser au fur et à mesure de la baisse du voltage de la batterie ;
- lorsque l'aimant est retiré, le pacemaker doit prendre pour hypothèse qu'il effectue un prétest ;
- ce mode doit pouvoir être désactivé, afin d'ignore la détection de l'aimant.

(21)/S (*Données implantatoires*) Le pacemaker doit pouvoir stocker les informations suivantes en mémoire :

- le modèle de pacemaker, numéro de série, date d'implantation ;
- la date d'implantation des sondes, leur polarité ;
- le seuil de stimulation, amplitude des signaux ;
- l'impédance des sondes ;
- le paramétrage des signaux spécifiques au patient.

3.3.4. Comportement du pacemaker

La section précédente a présenté les exigences liées au pacemaker, ainsi que ses différents modes de fonctionnement et son interaction avec le patient et l'équipe médicale. Nous avons mis en évidence les différents composants du pacemaker et les paramètres associés. Dans cette section, nous revenons sur le pacemaker, en étudiant son comportement temporel dans le cas de la thérapie de la bradycardie.

Le traitement de la bradycardie s'opère en stimulant le cœur du patient. Ce traitement doit être adapté au patient. Il y a donc plusieurs paramètres qui sont utilisés en fonction du mode de stimulation et des besoins du patient.

Nous définissons tout d'abord plusieurs termes qui seront utiles pour caractériser le comportement du pacemaker en réaction à une stimulation cardiaque, ou son absence passée un certain temps.

(22)/P (*Limite basse de stimulation*) La limite basse de stimulation (LRL) est le nombre de stimulations produites par minute (dans le cas atrial ou ventriculaire) en l'absence d'activité intrinsèque du cœur et de stimulation contrôlée par le capteur à une fréquence plus élevée.

La définition du LRL varie suivant le contexte :

- lorsque l'hystérésis est désactivée, la LRL définit la plus longue période autorisée entre deux stimulations ;

56 Systèmes embarqués

- dans les modes DXX ou VXX, la LRL démarre lorsqu'une pulsation du ventricule est détectée, ou qu'une stimulation est détectée ;
- dans les modes AXX, la LRL démarre lorsqu'une pulsation de l'oreillette est détectée, ou qu'une stimulation est détectée.

(23)/P (*Limite haute de stimulation*) La limite haute de pulsation (URL) est le nombre maximum d'événements mesurés au niveau de l'oreillette. L'URL est l'intervalle minimum entre un événement du ventricule et la prochaine stimulation du ventricule.

(24)/P (*Délai d'attente atrio-ventriculaire*) Le délai d'attente atrio-ventriculaire (AV) est un paramètre programmable représentant l'intervalle entre un événement atrial (intrinsèque ou stimulé) et une pulsation ventriculaire.

Dans les modes de suivi de l'oreillette, la stimulation ventriculaire doit se produire en absence d'un événement détecté du ventricule au plus tard après ce délai d'attente lorsque le taux de pulsation de l'oreillette est compris entre LRL et URL. Le délai d'attente peut être fixé (temps absolu) ou relatif.

Ce délai d'attente a une valeur différente suivant l'événement déclencheur :

(25)/P (*Délai d'attente atrio-ventriculaire stimulé*) Le délai d'attente stimulé doit avoir lieu lorsque le délai AV est initié par une stimulation de l'oreillette.

(26)/P (*Délai d'attente atrio-ventriculaire détecté*) Le délai d'attente détecté doit avoir lieu lorsque le délai AV est initié par une pulsation de l'oreillette.

(27)/P (*Délai d'attente atrio-ventriculaire dynamique*) Dans le cas où le délai d'attente est dynamique, sa valeur est déterminée pour chaque nouveau cycle cardiaque, en fonction des cycles précédents. Ce délai d'attente est obtenu en multipliant la durée du cycle précédent par une constante stockée en mémoire. Sa valeur est bornée par une valeur minimum et une valeur maximum.

(28)/P (*Décalage du délai d'attente*) Le paramètre configurable de décalage permet de réduire le délai d'attente après une détection d'une pulsation suivie au niveau de l'oreillette.

(29)/P (*Période réfractaire*) Afin de limiter les erreurs de détection, une période réfractaire pendant laquelle les mesures des sondes sont ignorées, peut être mise en place.

Différentes périodes réfractaires sont programmables, après un événement ventriculaire (VRP), atrial (ARP) et post atrio-ventriculaire (PVARP). Dans ce dernier cas, la période réfractaire est l'intervalle de temps suivant un événement ventriculaire lorsqu'un événement au niveau de l'oreillette n'inhibe pas une stimulation spontanée de l'oreillette ni ne déclenche une stimulation ventriculaire.

La période PVARP étendue est définie comme suit : si ce paramètre est actif, toute occurrence prématuée d'une contraction du ventricule (PVC) force une période réfractaire du générateur de pulsations pour une durée égale à ce paramètre ; la valeur de la période réfractaire doit revenir à la normale au cycle cardiaque suivant.

(30)/P (*Réponse au bruit*) En cas d'un bruit continu, le pacemaker doit être en mode de stimulation asynchrone.

(31)/P (*Réponse à une tachycardie atriale*) La réponse à une tachycardie atriale (ATR) vise à éviter un rythme cardiaque soutenu pendant de longues durées. Dans ce mode, le générateur de pulsations doit détecter une situation de tachycardie si le rythme intrinsèque de l'oreillette dépasse la valeur URL pendant un certain temps.

(32)/P *Détection de tachycardie* La détection de tachycardie fonctionne comme suit :

- le début d'une situation de tachycardie est détectée lorsque le rythme cardiaque est en moyenne supérieur à URL ;
- la fin d'une situation de tachycardie est détectée lorsque le rythme cardiaque est en moyenne inférieur à URL ;
- la période de détection doit être suffisamment courte pour que la thérapie se mette en place rapidement ;
- la période de détection doit être suffisamment longue pour que la thérapie ne se déclenche pas trop souvent, ni inutilement.

(33)/P (*ATR en cours*) Lorsqu'une situation de tachycardie est détectée, l'algorithme de l'ATR entre dans l'état « ATR en cours ». Le générateur de pulsations doit attendre un nombre de cycles cardiaques programmés avant d'entrer dans l'état « repli » (*fallback*). L'algorithme sort immédiatement de cet état lorsque l'attente est terminée, ou que la situation de tachycardie prend fin.

(34)/P (*Repli ATR*) Si la situation de tachycardie perdure après le délai d'attente précédent, alors (1) le générateur de pulsations entre dans l'état *Fallback* et passe en mode VVIR ; (2) le rythme de pulsations est réduit à LRL ; (3) ce mode prend fin dès que la situation de tachycardie cesse ; (4) le changement de mode est synchronisé sur une pulsation du ventricule ou un événement détecté.

(35)/P (*Adaptation du taux de stimulation*) Le pacemaker doit pouvoir ajuster le cycle cardiaque en fonction des besoins du métabolisme tels que mesurés par l'accéléromètre. Les paramètres suivants sont définis :

(36)/P (*Taux maximal « capteur »*) Le taux maximal « capteur » correspond à la valeur maximum du taux de stimulation en réponse aux mesures du capteur : (1) ce paramètre est requis pour les modes adaptatifs et (2) programmable indépendamment de l'URL.

(37)/P (*Seuil d'activité*) Ce seuil correspond au minimum d'activité détecté par l'accéléromètre avant une adaptation de la stimulation cardiaque.

(38)/P (*Facteur de réponse*) L'accéléromètre doit déterminer le nouveau taux de stimulation en fonction de l'état d'activité du patient. Plus forte est la valeur mesurée, plus rapide sera la réaction et la modification du taux de stimulation.

(39)/P (*Temps de réaction*) L'accéléromètre doit déterminer le taux d'augmentation du rythme de stimulation. Le temps de réaction correspond à la période pour passer de la valeur LRL à MSR.

(40)/P (*Temps de recouvrement*) L'accéléromètre doit déterminer le taux de recouvrement correspondant à la période pour passer de la valeur MSR à LRL, jusqu'à ce que l'activité passe sous le seuil d'activité.

(41)/P (*Stimulation en hystérésis*) Lorsque de mode est activé, les stimulations du générateur sont retardées afin de permettre une stimulation spontanée du cœur. Ce mode est actif lorsque le mode de stimulation est inhibé ou suivi, le rythme cardiaque est supérieur au taux limite d'hystérésis (HRL) ; lorsque le pacemaker est dans le mode AAI, un événement détecté au niveau de l'oreillette doit activer l'hystérésis ; ou enfin dans le mode inhibé et suivi avec stimulation du ventricule, un événement est détecté au niveau du ventricule.

(42)/P (*Lissage du rythme*) Les variations du rythme de stimulation sont lissées afin de ne pas être trop brusques. Ces paramètres sont donnés indépendamment pour l'augmentation et la diminution du taux.

3.4. Conclusion

Dans ce chapitre, nous avons présenté le cas d'étude commun, qui sera par la suite exploité en utilisant chacun des trois formalismes choisis dans cet ouvrage : SysML, MARTE et AADL.

Nous avons retenu comme cas d'étude un pacemaker, un système électrique dont le rôle est de stimuler le cœur d'un patient atteint de bradycardie. Nous avons détaillé la structure du pacemaker et son fonctionnement en présentant l'architecture générale du pacemaker et les exigences auxquelles il doit répondre telles qu'elles ont été rédigées par des experts.

Ce chapitre ne saurait être une référence complète sur le fonctionnement de ce dispositif. Son but est avant tout de fournir un vocabulaire de base qui servira aux autres chapitres. Néanmoins, nous avons fourni une traduction d'un document d'exigence,

illustrant ainsi une difficulté liée à l'ingénierie des systèmes embarqués : la communication de concepts métier à des non-spécialistes, dont la compétence est dans l'ingénierie logicielle.

Dans les chapitres suivants, les auteurs montreront comment exploiter ces spécifications pour chacun des formalismes retenus, et fourniront des compléments sur les spécifications fournies. En particulier sur les modes et le comportement du pacemaker, mais aussi son utilisation.

3.5. Bibliographie

- [BAR 10] BAROLD S., STROOBANDT R., SINNAEVE A., *Cardiac Pacemakers and Resynchronization Step by Step : An Illustrated Guide*, John Wiley & Sons, 2010.
- [Bos 07] BOSTON SCIENTIFIC, PACEMAKER System Specification, janvier 2007.
- [GOM 09] GOMES A. O., OLIVEIRA M. V., « Formal Specification of a Cardiac Pacing System », *Proceedings of the 2nd World Congress on Formal Methods*, FM '09, Berlin, Heidelberg, Springer-Verlag, p. 692–707, 2009.
- [MAC 08] MACEDO H., LARSEN P., FITZGERALD J., « Incremental Development of a Distributed Real-Time Model of a Cardiac Pacing System Using VDM », CUELLAR J., MAIBAUM T., SERE K., Eds., *FM 2008 : Formal Methods*, vol. 5014 de *Lecture Notes in Computer Science*, p. 181-197, Springer Berlin / Heidelberg, 2008.
- [MAN 09] MANNA V. P. L., BONANNO A. T., MOTTA A., A Simple Pacemaker Implementation, Rapport, 2009.
- [MÉR 09] MÉRY D., SINGH N. K., Pacemaker's Functional Behaviors in Event-B, Research Report, 2009.
- [NIX 09] NIXON C., ULRICH T., LARSON C., SMITH J., DAVIS R., CHA K., Academic Dual Chamber Pacemaker, Rapport, 2009.
- [TUA 10] TUAN L. A., ZHENG M. C., THO Q. T., « Modeling and Verification of Safety Critical Systems : A Case Study on Pacemaker », *Proceedings of the 2010 Fourth International Conference on Secure Software Integration and Reliability Improvement*, SSIRI '10, Washington, DC, USA, IEEE Computer Society, p. 23–32, 2010.
- [WOO 06] WOODCOCK J., « First Steps in the Verified Software Grand Challenge », *Computer*, vol. 39, p. 57-64, IEEE Computer Society, 2006.

DEUXIÈME PARTIE
SysML

Chapitre 4

Présentation des concepts de SysML

4.1. Introduction

Depuis longtemps, les ingénieurs système ont utilisé des techniques de modélisation. Parmi les plus connues, citons SADT et SA/RT, qui datent des années 1980, ainsi que de nombreuses approches basées sur les réseaux de Petri ou les machines à états finis. Mais ces techniques étaient limitées par leur portée et leur expressivité ainsi que par la difficulté de leur intégration avec d'autres formalismes.

L'essor d'UML dans le domaine du logiciel et l'effort industriel de développement d'outils qui l'accompagne ont naturellement conduit à envisager son utilisation en ingénierie système. Cependant, du fait de sa conception fortement guidée par les besoins du passage à la programmation par objets, le langage était, tout au moins dans ses premières versions, peu adapté à la modélisation des systèmes complexes et donc au support de l'ingénierie système.

La version 2 d'UML [OMG 05], officialisée en 2005, a introduit plusieurs nouveaux concepts et diagrammes utiles pour la modélisation système. En particulier, le diagramme de structure composite avec les concepts de classe structurée, partie, port et connecteur, permet maintenant de décrire l'interconnexion structurelle interne d'un système complexe. Les avancées du diagramme de séquence permettent également de décrire des scénarios d'interaction en ajoutant progressivement des niveaux d'architecture. Mais il reste toujours la barrière psychologique du vocabulaire orienté logiciel : classe, objet, héritage, etc.

Chapitre rédigé par Jean-Michel BRUEL et Pascal ROQUES.

La communauté de l'ingénierie système a voulu définir un langage commun de modélisation adapté à sa problématique, comme UML l'est devenu pour les informatiens. Ce nouveau langage, nommé SysML, est fortement inspiré de la version 2 d'UML, mais ajoute la possibilité de représenter les exigences du système, les éléments non logiciels (mécanique, hydraulique, capteur, etc.), les équations physiques, les flux continus (matière, énergie, etc.) et les allocations (voir section 4.7.2).

La version 1.0 du langage de modélisation SysML a été adoptée officiellement par l'OMG le 19 septembre 2007. Depuis, trois révisions mineures ont été publiées : SysML 1.1 en décembre 2008, SysML 1.2 en juin 2010 et SysML 1.3 en juin 2012. A l'heure où nous écrivons ce chapitre les outils implémentent la version 1.2 qui est donc celle retenue pour tous les diagrammes.

4.2. Origines de SysML

Le monde du logiciel a fini par se mettre d'accord à la fin des années 1990 sur un formalisme de modélisation unifié : UML. En 2003, l'INCOSE a décidé de faire d'UML ce langage commun pour l'IS. UML contenait en effet déjà à l'époque nombre de diagrammes indispensables, comme les diagrammes de séquence, d'états, de cas d'utilisation, etc. Le travail sur la nouvelle version UML2, entamé à l'OMG à peu près à la même période, a abouti à la définition d'un langage de modélisation très proche du besoin des ingénieurs système, avec des améliorations notables sur les diagrammes d'activité et de séquence, ainsi que la mise au point du diagramme de structure composite.

Cependant, il restait une barrière psychologique importante à l'adoption d'UML par la communauté de l'IS : sa teinture « logicielle » ! La possibilité d'extension d'UML, grâce au concept de stéréotype, a permis d'adapter le vocabulaire pour les ingénieurs système. En éliminant les mots « objet » et « classe » au profit du terme plus neutre « bloc », c'est-à-dire en gommant les aspects les plus informatiques d'UML, et en renommant ce langage de modélisation, l'OMG veut promouvoir SysML comme un nouveau langage différent d'UML, tout en profitant de sa filiation directe.

L'OMG a annoncé l'adoption de SysML en juillet 2006 et la disponibilité de la première version officielle (SysML v1.0) en septembre 2007. Une nouvelle spécification SysML v1.1 a été rendue publique en décembre 2008, et la révision courante SysML v1.3 a été publiée en juin 2012.

4.3. Présentation générale : les neuf types de diagrammes

UML 2.0 proposait treize types de diagrammes. Parmi ceux-ci certains ont été réutilisés tels quels, certains ont été modifiés et d'autres n'ont pas été retenus. SysML

s'articule autour de neuf types de diagrammes, que l'OMG a répartis en trois grands groupes (voir figure 4.1¹).

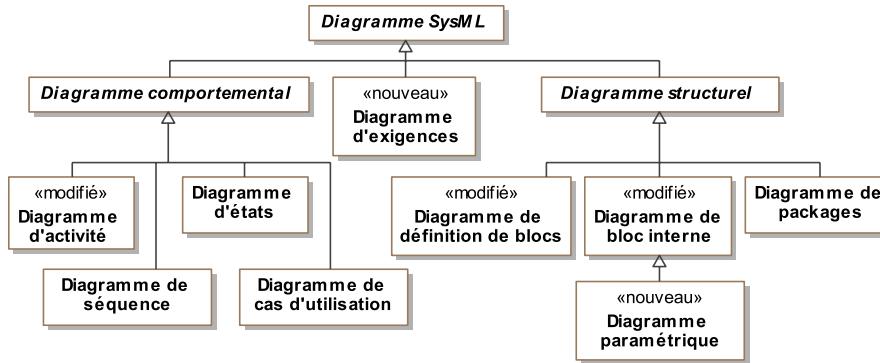


Figure 4.1 – Les neufs types de diagrammes SysML – traduit de [OMG 12]

Le premier groupe comprend quatre diagrammes comportementaux :

- diagramme d'activité (enchaînement des actions et décisions) ;
- diagramme de séquence (séquence verticale de messages) ;
- diagramme d'états (états et transitions possibles des blocs par exemple) ;
- diagramme de cas d'utilisation (fonctionnalités du système).

Le second groupe comprend un unique diagramme transverse : le diagramme d'exigences (montre les exigences du système et leurs relations).

Le troisième groupe comprend quatre diagrammes structurels :

- diagramme de définition de blocs (montre les briques de base statiques : blocs, compositions, associations, attributs, opérations, généralisations, etc.) ;
- diagramme de bloc interne (montre l'organisation interne d'un élément statique complexe, en termes de parties, ports, connecteurs, etc.) ;
- diagramme paramétrique (contraintes et équations) ;
- diagramme de *packages* (organisation logique du modèle).

1. La plupart des diagrammes de ce chapitre introductif sont issus du livre de P. Roques [ROQ 09] ou des documents de référence de l'OMG [OMG 12]. Les diagrammes de l'étude de cas traitée dans ce livre sont référencés dans les sections concernées.

La présentation officielle de l'organisation des différents types de diagrammes est donnée en figure 4.1. Cette figure détaille également les différences avec UML en indiquant les diagrammes modifiés ou repris tels quels.

Dans la suite de ce chapitre, nous présentons les types de diagrammes dans un ordre naturel d'utilisation : exigences, structure, dynamique et transverse.

4.4. Modélisation des exigences

La prise en compte des exigences en SysML peut se faire à plusieurs niveaux. SysML innove par rapport à UML en proposant un diagramme d'exigences qui permet de modéliser les exigences système et surtout de les relier ensuite aux éléments structurels ou dynamiques de la modélisation, ainsi qu'à des exigences de niveau sous-système. Nous commencerons néanmoins ce paragraphe par la description du diagramme de cas d'utilisation, déjà présent en UML, et qui est également très utile pour décrire les grandes fonctionnalités attendues du système et démarrer l'étude des exigences fonctionnelles.

4.4.1. Diagramme de cas d'utilisation

Le diagramme de cas d'utilisation (voir figure 4.2) est un schéma qui montre les cas d'utilisation (ovales) reliés par des associations (lignes) à leurs acteurs (icône du stickman, ou représentation graphique équivalente). Chaque association signifie simplement « participe à ». Ce type de diagramme est strictement identique à celui d'UML. Un cas d'utilisation (*use case*) représente un ensemble de séquences d'actions réalisées par le système et produisant un résultat observable intéressant pour un acteur particulier. Un acteur (*actor*) représente un rôle joué par un utilisateur humain ou un autre système qui interagit directement avec le système étudié (*subject*), représenté par le rectangle autour des cas d'utilisation.

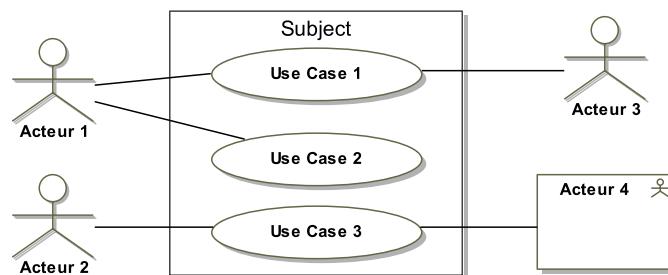


Figure 4.2 – Un diagramme de cas d'utilisation

Deux acteurs, ou plus, peuvent présenter des similitudes dans leurs relations aux cas d'utilisation. On peut l'exprimer en créant un acteur généralisé qui modélise les aspects communs aux différents acteurs concrets. Pour affiner encore le diagramme de cas d'utilisation, SysML définit trois types de relations standardisées entre cas d'utilisation (voir figure 4.3) :

- une relation d'inclusion, formalisée par un mot-clé *include* : le cas d'utilisation de base en incorpore explicitement un autre, de façon obligatoire ;
- une relation d'extension, formalisée par un mot-clé *extend* : le cas d'utilisation de base en incorpore implicitement un autre, de façon optionnelle, à un endroit spécifié indirectement dans celui qui procède à l'extension (appelé *extension point*) ;
- une relation de généralisation/specialisation (flèche blanche) : les cas d'utilisation descendants héritent la description de leur parent commun. Chacun d'entre eux peut néanmoins comprendre des interactions spécifiques supplémentaires.

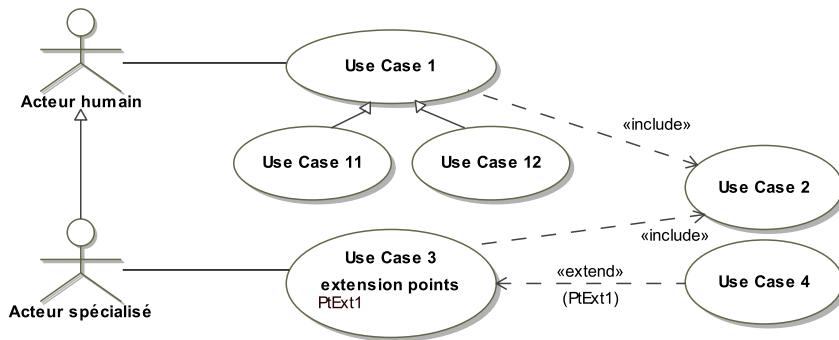


Figure 4.3 – Concepts avancés du diagramme de cas d'utilisation

Pour une illustration concrète des diagrammes de cas d'utilisation de l'étude de cas du livre, voir figures 5.3 et 8.1.

4.4.2. Diagramme d'exigences

Le diagramme d'exigences permet de représenter graphiquement les exigences dans le modèle. Les deux propriétés de base d'une exigence (*requirement*) sont d'une part un identifiant unique (permettant ensuite de gérer la traçabilité avec l'architecture, etc.), et d'autre part un texte descriptif (voir figure 4.4).

Il est courant de définir d'autres propriétés pour les exigences. La liste ci-dessous n'est donc pas exhaustive :

- priorité (par exemple, haute, moyenne, basse) ;

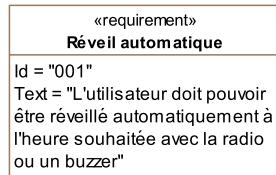


Figure 4.4 – Exemple d'exigence [ROQ 09]

- source (par exemple, client, marketing, technique, législation, etc.) ;
- risque (par exemple, haut, moyen, bas) ;
- statut (par exemple, proposée, validée, implémentée, testée, livrée, etc.) ;
- méthode de vérification (par exemple, analyse, démonstration, test, etc.).

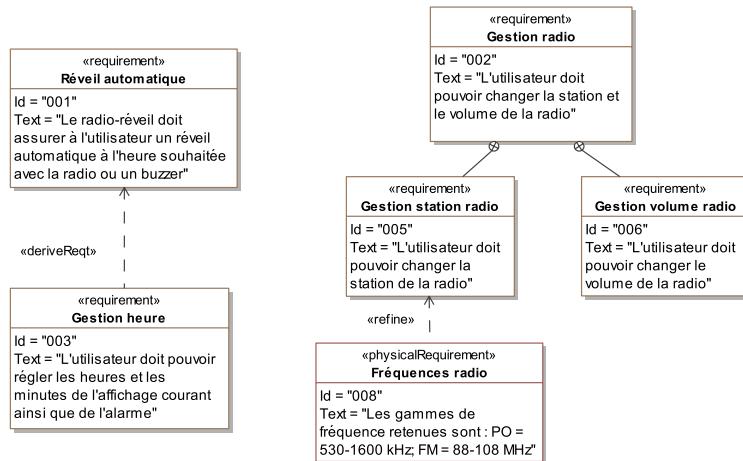


Figure 4.5 – Exemple de diagramme d'exigences [ROQ 09]

Les exigences peuvent être reliées entre elles par des relations de contenance, de raffinement ou de dérivation (voir figure 4.5) :

- la contenance (ligne terminée par un cercle contenant une croix du côté du conteneur) permet de décomposer une exigence composite en plusieurs exigences unitaires, plus faciles ensuite à tracer vis-à-vis de l'architecture ou des tests ;
- le raffinement (*refine*) consiste en l'ajout de précisions : par exemple des données quantitatives ;

– la dérivation (*deriveReqt*) consiste à relier des exigences de niveaux différents : par exemple des exigences système à des exigences de niveau sous-système, etc. Elle implique généralement des choix d'architecture.

Le diagramme d'exigences permet ensuite tout au long d'un projet de relier les exigences avec d'autres types d'élément SysML par plusieurs relations² (voir figure 4.6) :

- la relation entre une exigence et un élément comportemental (cas d'utilisation, diagramme d'états, etc.) se note par le mot-clé *refine* ;
- la relation entre une exigence et un bloc d'architecture se note par le mot-clé *satisfy* ;
- la relation entre une exigence et un cas de test se note par le mot-clé *verify*.

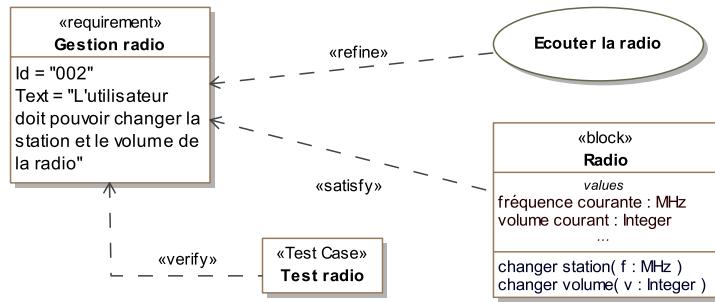


Figure 4.6 – Exemple de relations avec les exigences [ROQ 09]

Pour une illustration concrète des diagrammes d'exigence de l'étude de cas du livre, voir les figures 5.5, 5.14, 5.17 et 5.20.

4.5. Modélisation structurelle

Pour la modélisation des aspects statiques du système, SysML propose deux principaux types de diagrammes : le diagramme de définition de blocs (*bdd*) et le diagramme interne de blocs (*ibd*). Le bloc SysML (*block*) constitue la brique de base pour la modélisation de la structure d'un système. On peut s'en servir pour représenter des entités physiques (système complet, sous-système ou composant élémentaire), mais aussi des entités logiques ou conceptuelles. Le bloc permet de décrire également les flots (de données et/ou de contrôle) qui circulent à travers un système. Les blocs sont décomposables et peuvent posséder un comportement.

2. Une autre représentation graphique proposée par SysML consiste à faire apparaître les relations de traçabilité par une note attachée à l'exigence, ou encore par des attributs ou des compartiments supplémentaires. Une autre possibilité consiste à utiliser une forme tabulaire.

4.5.1. Diagramme de définition de blocs

Le diagramme de définition de blocs (*block definition diagram*) est utilisé pour représenter les blocs, leurs propriétés, leurs relations. Dans un *bdd*, un bloc est représenté graphiquement par un rectangle découpé en compartiments (voir figure 4.7). Le nom du bloc apparaît tout en haut, et constitue l’unique compartiment obligatoire. Tous les autres compartiments ont des labels indiquant ce qu’ils contiennent : valeurs, parties, etc. Les propriétés sont les caractéristiques structurelles de base des blocs. Elles peuvent être de plusieurs types :

- les valeurs (*value properties*) décrivent des caractéristiques quantifiables en terme de *value types* (domaine de valeur, dimension et unité optionnelle) ;
- les parties (*part properties*) décrivent la hiérarchie de décomposition du bloc en termes d’autres blocs ;
- les références (*reference properties*) décrivent des relations de type association ou simple agrégation avec d’autres blocs.

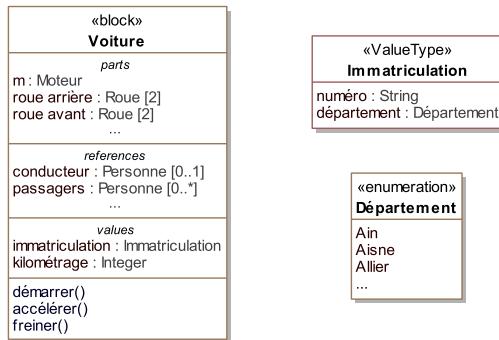


Figure 4.7 – Notation du bloc dans le *bdd*

Pour typer les valeurs (*value properties*), SysML propose de définir des *Value Types*. Dans l’exemple de la figure 4.7, la *value types* *Immatriculation* permet de créer un type que l’on réutilise par la suite dans le bloc *Voiture*.

Il existe deux principaux types de relations entre blocs : l’association (avec ses cas particuliers que sont l’agrégation et la composition) et la généralisation (voir figure 4.8).

L’association est une relation statique, durable, entre deux blocs. A chacune de ses deux extrémités doit figurer une indication de multiplicité. Elle spécifie, sous la forme d’un intervalle, le nombre d’instances³ qui peuvent participer à une relation avec une

3. Une instance est un exemplaire d’un certain bloc possédant une identité propre.

instance de l'autre bloc dans le cadre de cette association. Une association unidirectionnelle possède une flèche pointant vers le bloc référencé.

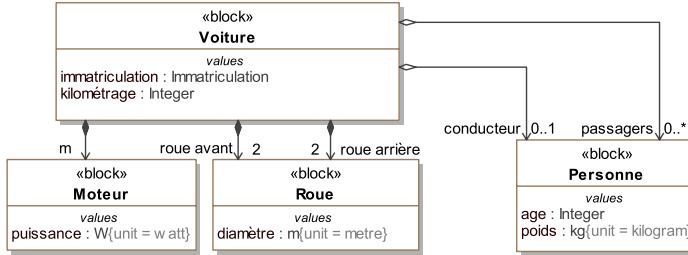


Figure 4.8 – Exemples de relations dans le *bdd*

Quand l'association possède un caractère de « contenance », on utilise la relation d'agrégation/composition. La relation de composition entre blocs, dans laquelle l'un des blocs représente le tout, et les autres ses parties, peut être représentée graphiquement par un losange plein du côté du conteneur. Par exemple, la suppression d'une instance du conteneur va entraîner en cascade la suppression des instances contenues. La relation d'agrégation (losange vide) est beaucoup moins forte que la relation de composition (losange plein) au sens où les instances peuvent exister indépendamment entre elles.

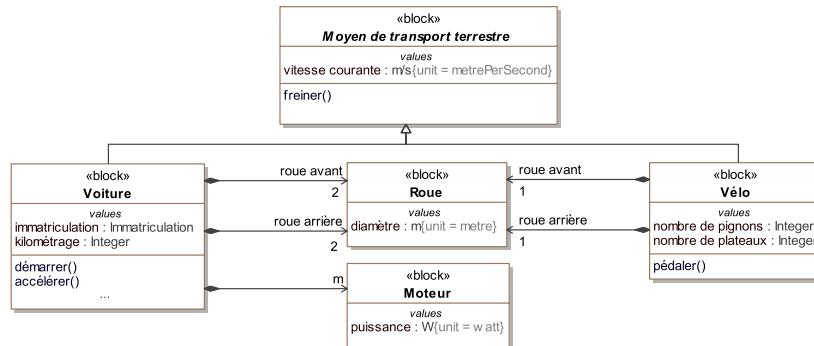


Figure 4.9 – Exemples de généralisation dans le *bdd*

Les blocs peuvent être organisés dans une hiérarchie de classification. Le but est souvent de factoriser des propriétés communes à plusieurs blocs (valeurs, parties, etc.) à l'intérieur d'un bloc généralisé. Les blocs spécialisés « héritent » des propriétés du bloc généralisé et peuvent posséder des propriétés supplémentaires. La

généralisation se représente graphiquement par une flèche triangulaire pointant sur le bloc généralisé (voir figure 4.9).

Pour une illustration concrète des diagrammes de définition de blocs de l'étude de cas du livre, voir les figures 5.1, 5.9, 5.10 et 5.11.

4.5.2. Diagramme interne de bloc

Le diagramme interne de blocs (*internal block diagram* ou *ibd*) décrit la structure interne d'un bloc en termes de parties, ports et connecteurs. Il est important de noter que l'on peut représenter plusieurs niveaux de décomposition sur un même *ibd*.

Une relation de composition dans un *bdd* peut se représenter au travers d'un *ibd*. Le cadre de l'*ibd* représente alors le bloc englobant. Il fournit le contexte pour tous les éléments du diagramme. Chaque extrémité de la relation de composition du *bdd* apparaît comme un bloc (appelé partie) à l'intérieur du cadre dans l'*ibd*. Le nom de la partie est de la forme : nom_partie : nom_bloc [multiplicité] (voir figure *figIBD*). La multiplicité (1 par défaut) peut également être représentée dans le coin supérieur droit du rectangle. Les associations et les agrégations extérieures au bloc englobant sont représentées de façon similaire aux compositions, sauf que le rectangle du bloc est en pointillés.

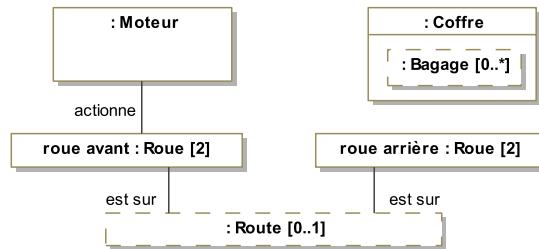


Figure 4.10 – Diagramme interne de blocs

Le connecteur est un concept structurel utilisé pour relier deux parties et leur fournir l'opportunité d'interagir (le terme « actionne » dans la figure 4.10). L'extrémité d'un connecteur peut posséder une multiplicité qui décrit le nombre d'instances qui peuvent être connectées par des liens décrits par le connecteur.

Le diagramme interne de blocs permet également de décrire la logique de connexion, de services et de flots entre blocs grâce au concept de port. Les ports définissent les points d'interaction offerts (*provided*) et requis (*required*) entre les blocs. Un bloc peut avoir plusieurs ports qui spécifient des points d'interaction différents. Les ports peuvent être de deux natures :

- standard : ce type de port autorise la description de services logiques entre les blocs, au moyen d’interfaces regroupant des opérations (il s’agit de la notion classique d’API en programmation). Ils sont représentés par des carrés vides ;
- flux (*flow port*) : ce type de port, nouveau en SysML, permet de représenter la circulation de flux physiques entre blocs. La nature de ce qui peut circuler va des fluides aux données, en passant par l’énergie.

Les ports de type « flux » sont soit atomiques (un seul flux), soit composites (agrégation de flux de natures différentes). Un *flow port* atomique ne spécifie qu’un seul type de flux en entrée ou en sortie (ou les deux), la direction étant simplement indiquée par une flèche à l’intérieur du carré représentant le port. Il peut être typé par un bloc ou un *Value Type* représentant le type d’élément pouvant circuler en entrée ou en sortie du port (voir figure 4.11).

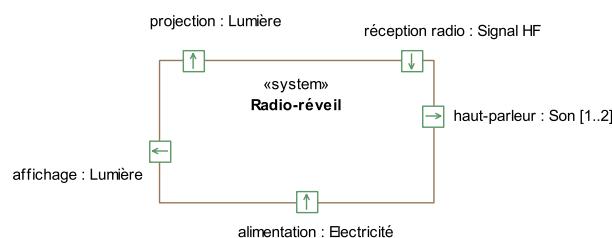


Figure 4.11 – *Flow Ports* atomiques [ROQ 09]

Quand un point d’interaction a une interface complexe avec plusieurs flux, le *flow port* correspondant doit être modélisé comme un *flow port* composite (ou non atomique). Dans ce cas, le port doit être typé par une spécification de flux (*Flow Specification*). Cette spécification de flux doit être définie dans un *bdd*. Elle inclut plusieurs propriétés de flux, chacune ayant un nom, un type et une direction. Un *flow port* composite est indiqué graphiquement par deux crochets se faisant face (<>) dessinés à l’intérieur du symbole du port. Le *flow port* conjugué (préfixé par « ~ » ou bien représenté en noir) prend les entrées/sorties inverses de la spécification de flux (voir figure 4.12).

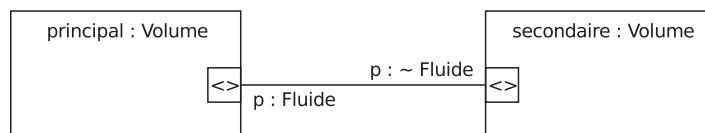


Figure 4.12 – *Flow ports* non atomiques et conjugués

Une interface fournie (*provided interface*) sur un port standard spécifie les opérations que le bloc fournit. La relation de réalisation entre un bloc et une interface se dessine

comme une généralisation en pointillés. Si l'interface est représentée par un simple cercle, la réalisation devient un simple trait (voir figure 4.13). Une interface requise (*required interface*) spécifie les opérations dont le bloc a besoin pour réaliser son comportement. La relation d'utilisation entre un bloc et une interface se dessine comme une flèche évidée en pointillés. Dans la notation graphique condensée, l'utilisation devient un simple trait et l'interface est représentée par un demi-cercle (voir figure 4.13).

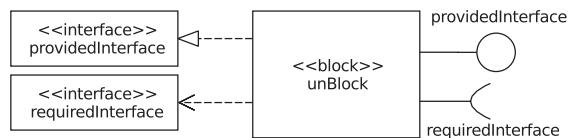


Figure 4.13 – Port standard avec interfaces fournies et requises

Pour une illustration concrète des diagrammes interne de blocs de l'étude de cas du livre, voir figure 5.12.

4.5.3. Diagramme de packages

Le diagramme de *packages*⁴ montre l'organisation logique du modèle sous forme d'arborescence ainsi que les éventuelles relations de dépendance entre les packages. Un package constitue un espace de noms (*namespace*) pour les éléments qu'il contient. Il y a ainsi deux types de relations possibles entre *packages* : la contenance et la dépendance.

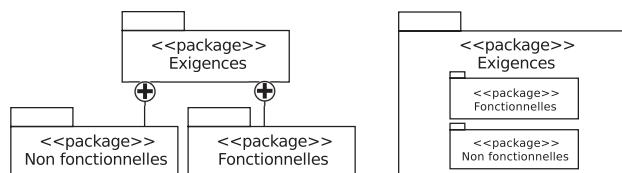


Figure 4.14 – Contenance entre packages

La relation de contenance peut être représentée de deux façons :

- une croix entourée du côté du conteneur (⊕);
- une imbrication graphique des *packages* contenus à l'intérieur du *package* englobant (voir figure 4.14).

4. Nous n'utilisons volontairement pas le terme de « paquetage » parfois rencontré dans la littérature francophone.

Selon l'organisation du modèle et les choix de structuration, les éléments de différents packages sont souvent reliés entre eux. Nous avons vu par exemple que les blocs peuvent être reliés par des associations, des compositions, des généralisations, etc. Ces relations entre éléments induisent des relations de dépendance entre les packages englobants, représentées par des flèches pointillées (voir figure 4.15).

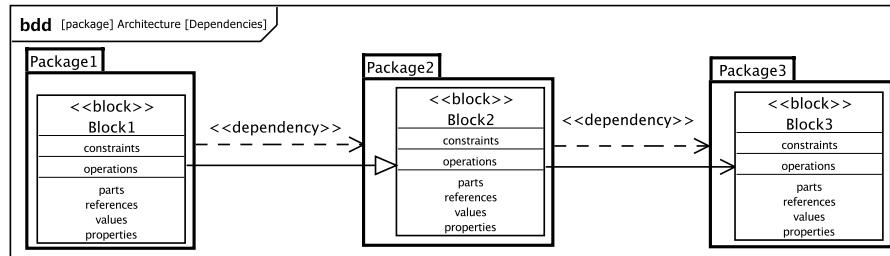


Figure 4.15 – Relations entre blocs et dépendance entre packages

Une vue (*view*) est une sorte de *package* utilisée pour montrer une perspective particulière sur un modèle, comme la sécurité, ou les performances (voir figure 4.16). Une vue est conforme à un point de vue.

Un point de vue (*viewpoint*) représente une perspective particulière qui spécifie le contenu d'une vue. Un point de vue contient des propriétés standardisées par SysML, comme illustré sur l'exemple en figure 4.16 :

- *concerns* : les préoccupations des parties prenantes ;
- *languages* : les langages utilisés pour présenter la vue ;
- *methods* : les méthodes utilisées pour établir la vue ;
- *purpose* : la raison de la présentation de cette vue ;
- *stakeholders* : les parties prenantes ayant un intérêt dans la vue.

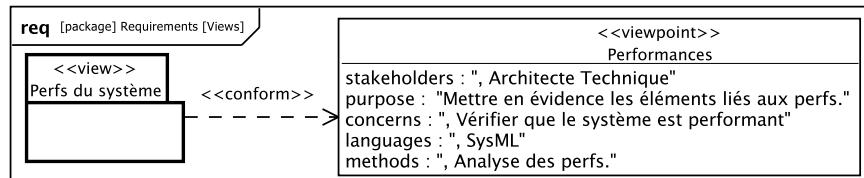


Figure 4.16 – La représentation graphique du *viewpoint* [ROQ 09]

Pour une illustration concrète des diagrammes de *package* de l'étude de cas du livre, voir figure 7.1.

4.6. Modélisation dynamique

Les diagrammes regroupés dans cette section concernent les aspects comportementaux du système. Le diagramme des cas d'utilisation est classé dans cette rubrique dans la documentation de l'OMG mais a été présenté précédemment (voir section 4.4). Trois diagrammes complémentaires permettent de modéliser le comportement du système : le diagramme de séquence modélise les échanges entre éléments au sein d'une interaction ; le diagramme d'états permet de modéliser le comportement d'un élément sous forme d'états et de transitions ; et le diagramme d'activité permet d'exprimer la logique de contrôle et d'entrées/sorties d'un traitement complexe.

4.6.1. Diagramme de séquence

Le diagramme de séquence SysML est repris tel quel d'UML 2. Il montre la séquence verticale des messages passés entre éléments (lignes de vie) au sein d'une interaction.

Une ligne de vie possède un nom et un type. Elle est représentée graphiquement par une ligne verticale en pointillés.

Un message représente une communication unidirectionnelle entre lignes de vie qui déclenche une activité dans le destinataire. Un message synchrone (émetteur bloqué en attente de réponse) est représenté par une flèche pleine, alors qu'un message asynchrone est représenté par une flèche évidée. La flèche qui boucle (message réflexif) permet de représenter un comportement interne. La flèche pointillée représente un retour. Cela signifie que le message en question est le résultat direct du message précédent (voir figure 4.17).

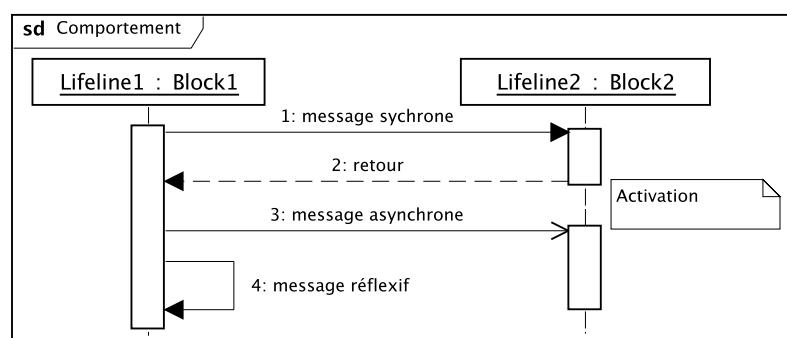


Figure 4.17 – Notation de base du diagramme de séquence [ROQ 09]

SysML propose une notation très utile : le fragment combiné. Chaque fragment possède un opérateur et peut être divisé en opérandes. Les principaux opérateurs sont :

- *loop* : boucle. Le fragment peut s'exécuter plusieurs fois, et la condition de garde explicite l'itération ;
- *opt* : optionnel. Le fragment ne s'exécute que si la condition fournie est vraie ;
- *alt* : fragments alternatifs. Seul le fragment dont la condition s'évalue à vraie s'exécutera (voir figure 4.18).

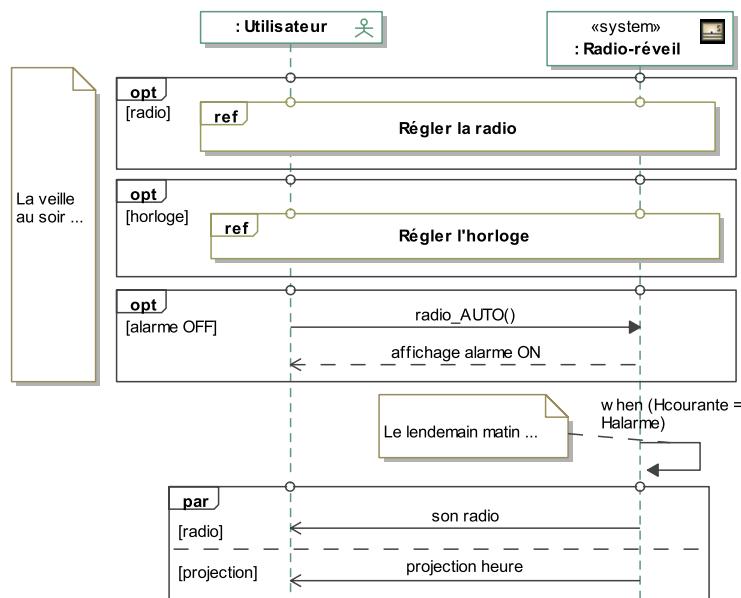


Figure 4.18 – Exemple de fragment combiné [ROQ 09]

Un diagramme de séquence peut aussi en référencer un autre grâce à un rectangle avec le mot-clé *ref*. Cette notation est très pratique pour modulariser les diagrammes de séquence, et créer des hyperliens graphiques exploitables par les outils de modélisation.

SysML permet encore d'ajouter des contraintes temporelles sur le diagramme de séquence. Il existe deux types de contraintes :

- la contrainte de durée, permet d'indiquer une contrainte sur la durée exacte, la durée minimum ou la durée maximum entre deux événements ;
- la contrainte de temps, permet de positionner des labels associés à des instants dans le scénario au niveau de certains messages et de les relier ainsi entre eux (voir figure 4.19).

Pour une illustration concrète des diagrammes de séquence de l'étude de cas du livre, voir les figures 5.4 et 5.18.

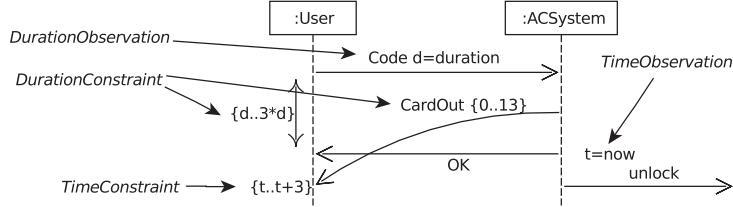


Figure 4.19 – Exemple de contraintes temporelles [OMG 12]

4.6.2. Diagramme d'états

SysML a repris le concept bien connu de machine à états finis (déjà présent dans UML), qui consiste à s'intéresser au cycle de vie d'une instance générique d'un bloc particulier (au sens décrit dans la partie structurelle précédente) dans tous les cas possibles de ses interactions (avec les autres blocs). Cette vue locale d'un bloc, qui décrit comment il réagit à des événements en fonction de son état courant et comment il passe dans un nouvel état, est représentée graphiquement sous la forme d'un diagramme d'états.

Un état représente une situation durant la vie d'un bloc pendant laquelle :

- il satisfait une certaine condition ;
- il exécute une certaine activité ;
- il attend un certain événement.

Un bloc passe par une succession d'états durant son existence. Un état a une durée finie, variable selon la vie du bloc, en particulier en fonction des événements qui lui arrivent.

Une transition décrit la réaction d'un bloc lorsqu'un événement se produit (généralement le bloc change d'état, mais pas forcément). En règle générale, une transition possède un événement déclencheur, une condition de garde, un effet et un état cible. Une transition peut spécifier un comportement optionnel réalisé par le bloc lorsque la transition est déclenchée. Ce comportement est appelé « effet » : cela peut être une simple action ou une séquence d'actions. Les activités durables (*do-activity*) ont une certaine durée, sont interruptibles et sont associées aux états.

En plus de la succession d'états « normaux » correspondant au cycle de vie d'un bloc, le diagramme d'états comprend également deux pseudo-états :

- l'état initial du diagramme d'états correspond à la création d'une instance ;
- l'état final du diagramme d'états correspond à la destruction de l'instance.

La notation de base du diagramme d'états est donnée dans la figure 4.20.

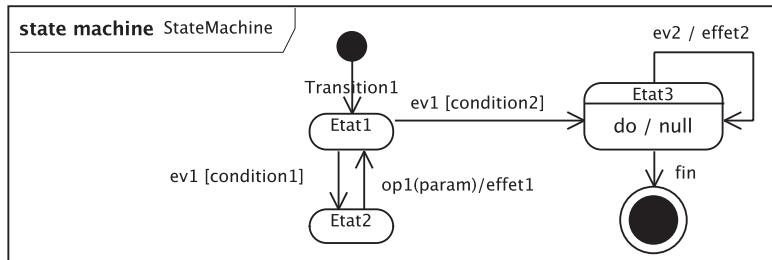


Figure 4.20 – Notation de base du diagramme d'états [ROQ 09]

Un état composite (aussi appelé super-état) permet d'englober plusieurs sous-états exclusifs. On peut ainsi factoriser des transitions déclenchées par le même événement et amenant vers le même état cible, tout en spécifiant des transitions particulières entre les sous-états. Une autre façon de représenter un état composite consiste à ajouter un symbole en forme d'haltère en bas à droite du rectangle à coins arrondis, puis à décrire les transitions entre ses sous-états dans un autre diagramme. On peut ainsi faire de la décomposition hiérarchique d'états, en gardant chaque niveau lisible et relativement simple. On peut même réutiliser des machines à états décrites par ailleurs.

Un état composite peut également contenir des régions concurrentes, il suffit graphiquement de le séparer par des traits pointillés. Chaque région peut alors être nommée (optionnel). Elle contient ses propres états et ses propres transitions. Les régions sont dites concurrentes car elles peuvent évoluer en parallèle et indépendamment.

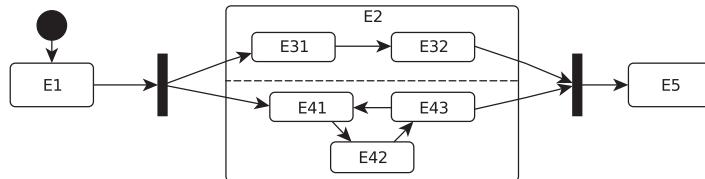


Figure 4.21 – Exemple de régions concurrentes

Le formalisme du diagramme d'états SysML contient encore de nombreux autres concepts avancés que nous ne présenterons pas dans cette introduction car ils sont beaucoup moins utilisés (*entry*, *exit*, transition interne, etc.).

Pour une illustration concrète des diagrammes d'état de l'étude de cas du livre, voir les figures 5.2, 5.13 et 5.15.

4.6.3. Diagramme d'activité

Le diagramme d'activité représente les flots de données et de contrôle entre les actions. Il s'agit d'un diagramme déjà présent dans UML 2 avec quelques ajouts, dont le plus important concerne la modélisation des flots continus.

Il est utilisé majoritairement pour l'expression de la logique de contrôle et d'entrées/-sorties. Les éléments de base du diagramme d'activité sont les suivants :

- des actions ;
- des flots de contrôle entre actions ;
- des décisions (aussi appelées branchements conditionnels) ;
- un début et une ou plusieurs fins possibles.

L'action est l'unité fondamentale de spécification comportementale en SysML. Elle représente un traitement ou une transformation. Les actions sont contenues dans les activités, qui fournissent leur contexte. Un flot est un contrôle de séquençage pendant l'exécution de nœuds d'activité. Les flots de contrôle sont de simples flèches reliant deux nœuds (actions, décisions, etc.). Le diagramme d'activité permet également d'utiliser des flots d'objets (reliant une action et un objet consommé ou produit). Une décision est un nœud de contrôle structuré représentant un choix dynamique entre plusieurs conditions qui doivent être mutuellement exclusives. Elle est représentée par un losange qui possède un arc entrant et plusieurs arcs sortants (voir figure 4.22).

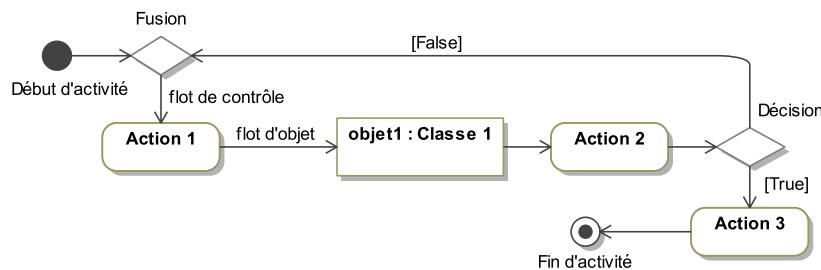


Figure 4.22 – Les bases du diagramme d'activité

Un *fork* est un nœud de contrôle structuré représentant un débranchement parallèle. Il est représenté par une barre horizontale ou verticale qui possède un arc entrant et plusieurs arcs sortants. Le *fork* duplique le jeton entrant sur chaque flot sortant. Les jetons sur les arcs sortants sont indépendants et concurrents. Un *join* est un nœud de contrôle structuré représentant une synchronisation entre actions (rendez-vous). Il est représenté par une barre horizontale ou verticale qui possède un arc sortant et plusieurs arcs entrants. Le *join* ne produit son jeton de sortie que lorsqu'un jeton est disponible sur chaque flot entrant.

Un autre nœud de contrôle intéressant est la fin de flot (*flow final*). Contrairement à la fin d'activité qui est globale à l'activité, la fin de flot est locale au flot concerné et n'a pas d'effet sur l'activité englobante.

Il existe un dernier nœud de contrôle : la fusion (*merge*). C'est l'inverse de la décision : le même symbole du losange, mais cette fois-ci avec plusieurs flots entrants et un seul sortant.

Le diagramme d'activité sert non seulement à préciser la séquence d'actions à réaliser, mais aussi ce qui est produit, consommé ou transformé au cours de l'exécution de cette activité.

Pour cela, SysML propose les concepts de flot d'objet (*object flow*) et de broche d'entrée ou sortie (*input/output pin*). Une action traite les jetons placés sur ses broches d'entrée. Ces jetons, qui peuvent être typés par des blocs, sont consommés ou transformés par l'action, puis placés sur les broches de sortie pour alimenter d'autres actions.

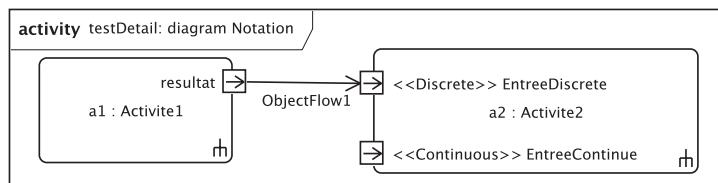


Figure 4.23 – Actions avec broches, flot d'objet et nœud d'objet

Les activités peuvent être réutilisées à travers des actions d'appel (*callBehaviorAction*). L'action d'appel est représentée graphiquement par une fourche à droite de la boîte d'action (voir figure 4.23), ainsi que par la chaîne : nom d'action : nom d'activité. SysML propose encore bien d'autres concepts et notations, comme la région interruptible, la région d'expansion ou encore les flots de type *stream*.

Pour permettre la modélisation de systèmes continus, SysML ajoute à UML 2 la possibilité de caractériser la nature du débit qui circule sur le flot : continu (par exemple, courant électrique, fluide, etc.) ou discret (par exemple, événements, requêtes, etc.). On utilise pour cela des stéréotypes : *continuous* et *discrete* (voir figure 4.23). Par défaut, un flot est supposé discret.

Pour une illustration concrète des diagrammes d'activité de l'étude de cas du livre, voir les figures 5.7 et 5.8.

4.7. Modélisation transverse

Indépendamment du caractère structurel ou comportemental, SysML ajoute à UML 2 un certain nombre de concepts généraux.

SysML permet d'utiliser les notes graphiques d'UML sur tous les types de diagrammes (forme de post-it). Deux mots-clés particuliers ont été ajoutés afin de représenter (voir figure 4.24) :

- des problèmes à résoudre (*problem*) ;
- des justificatifs (*rationale*).

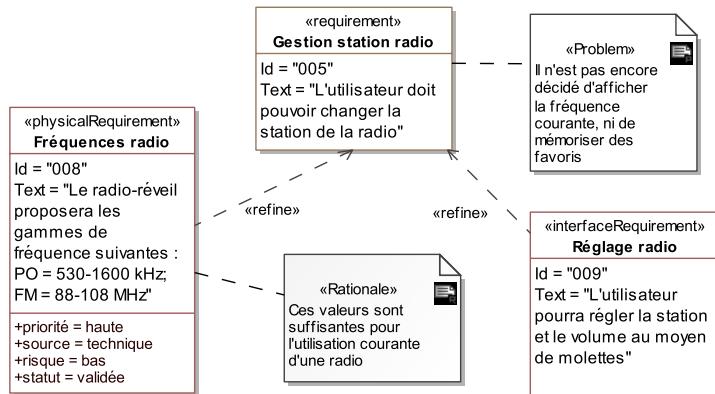


Figure 4.24 – Notes prédéfinies pour problèmes et justificatifs en SysML [ROQ 09]

4.7.1. Diagramme paramétrique

Le diagramme paramétrique permet de représenter des contraintes sur les valeurs de paramètres système tels que performance, fiabilité, masse, etc. Ce nouveau diagramme fournit ainsi un support précieux pour les études d'analyse système.

Chaque contrainte est d'abord définie par des paramètres ainsi qu'une règle décrivant l'évolution des paramètres les uns par rapport aux autres. Une contrainte est représentée par un bloc avec un stéréotype *constraint*. Il faut donc déclarer les contraintes dans un *bdd*, comme pour les blocs plus classiques (voir figure 4.25).

Les contraintes s'appuient ensuite sur le diagramme paramétrique, qui est une spécialisation du diagramme de bloc interne, pour permettre leur composition et leur mise en relation. Les *constraint properties* sont représentées différemment des *part properties* de l'*ibd* : ce sont des rectangles aux coins arrondis. Les paramètres formels des

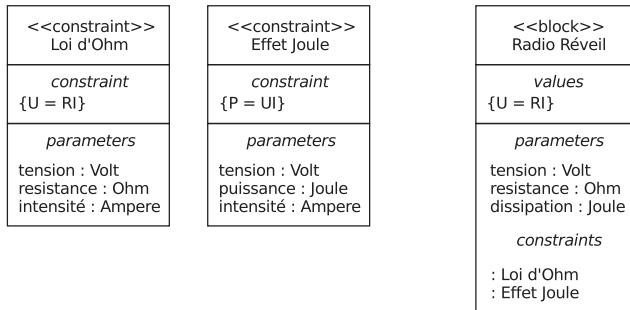


Figure 4.25 – Exemple de déclaration de contraintes [ROQ 09]

équations sont représentés par des ports et peuvent ainsi être connectés les uns aux autres (voir figure 4.26).

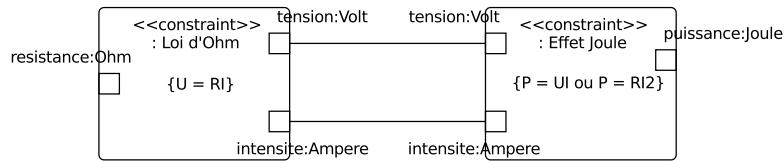


Figure 4.26 – Exemple de réseau de contraintes [ROQ 09]

La contrainte s’instancie ensuite en reliant des valeurs appartenant aux blocs du modèle aux paramètres formels : c’est la notion de *value binding*.

4.7.2. Allocation et traçabilité

L’allocation est un mécanisme général en ingénierie système pour interconnecter des éléments de différents types. Par exemple, pour relier un flot d’objet dans un diagramme d’activité à un connecteur dans un diagramme de bloc interne (*ibd*), ou une action à un bloc, etc. Ces relations sont souvent présentes dans un modèle classique, mais uniquement sous forme de commentaires ou de notes, donc pas très visibles. Le gros apport de SysML avec le mécanisme d’allocation est de rendre visible et exploitable toutes ces interconnexions.

Comme le mécanisme d’allocation est très général, SysML n’a pas ajouté un diagramme spécifique, mais plutôt un mécanisme avec plusieurs représentations graphiques disponible dans de nombreux diagrammes.

La première représentation graphique est donnée par une flèche pointillée avec le mot-clé *allocate* (voir figure 4.27).

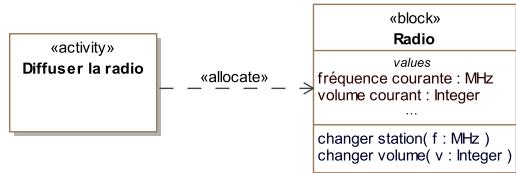


Figure 4.27 – Notation générale de l’allocation [ROQ 09]

Une seconde façon consiste à montrer les relations dans un compartiment supplémentaire. L’élément pointé par la relation *allocate* possède une propriété *allocatedFrom*, l’élément cible possède une propriété *allocatedTo* (voir figure 4.28).



Figure 4.28 – Notation de l’allocation par ajout de compartiments [ROQ 09]

Une troisième façon de représenter ces propriétés consiste à les faire figurer dans une note attachée (voir figure 4.29).

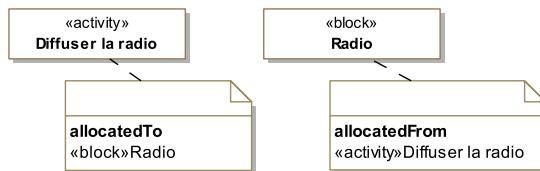


Figure 4.29 – Notation de l’allocation par ajout de notes [ROQ 09]

Comme c'est également le cas pour les exigences, SysML préconise de pouvoir représenter les allocations sous forme tabulaire. Pour une illustration concrète d’allocations tirées de l’étude de cas du livre, voir les figures 5.16 et 5.19.

4.8. Environnement et outillage

Parmi les principaux outils actuels permettant la modélisation SysML, citons : Atego / ARTiSAN Studio, IBM / Rhapsody, No Magic / MagicDraw, MODELIOSOFT / Modelio, Sparx Systems / Enterprise Architect, Papyrus (*open source*), Topcased (*open source*).

Il est à noter que cette liste d'outils évolue régulièrement. Leurs fonctionnalités détaillées (édition, simulation, génération de documentation, passerelles vers d'autres langages et outils, etc.) sont également mises à jour fréquemment⁵.

4.9. Conclusion

SysML se définit comme un langage de modélisation pour l'analyse et la spécification des systèmes complexes. Il est basé sur UML et en réutilise certains diagrammes importants. Si UML est très bien adapté pour l'ingénierie du logiciel, il montre cependant des lacunes pour l'ingénierie des systèmes comme par exemple la modélisation des exigences ou des flux continus.

En réduisant le nombre de diagrammes proposés par UML, tout en étendant néanmoins sa capacité de modélisation, SysML vise à adresser l'éventail complet de la problématique de l'ingénierie des systèmes.

4.10. Bibliographie

- [FRI 11] FRIEDENTHAL S., MOORE A., STEINER R., *A Practical Guide to SysML, 2nd edition*, OMG Press, 2011.
- [HOL 07] HOLT J., PERRY S., *SysML for Systems Engineering : The Emperor's New Modeling Language*, IET, 2007.
- [OMG 05] OMG, Unified Modeling Language Specification 2.0 : Superstructure, 2005, OMG doc. formal/05-07-04.
- [OMG 12] OMG, Site officiel de l'OMG, www.omg.org/sysml/, 2012.
- [ROQ 09] ROQUES P., *SysML par l'exemple*, Eyrolles, 2009.
- [WEI 08] WEILKENS T., *Systems Engineering with SysML/UML : Modeling, Analysis, Design*, OMG Press, 2008.

5. Pour une illustration des fonctionnalités comparées de certains de ces outils, voir fr.wikipedia.org/wiki/Comparaison_des_logiciels_d'UML.

Chapitre 5

Modélisation de l'étude de cas avec SysML

5.1. Introduction

Dans ce chapitre, nous allons construire le modèle SysML de notre étude de cas à partir des exigences définies au chapitre 3.

Ce travail se déroule en quatre phases selon la chronologie d'activités suivante :

1) la spécification du système (détaillée en section 5.2) : il s'agit ici de décrire le contexte, de reprendre dans le modèle les exigences textuelles exprimées précédemment, d'identifier les cas d'utilisation du système représentant ses capacités principales, enfin d'établir la traçabilité entre les exigences et les cas d'utilisation ;

2) la conception du système (détaillée en section 5.3) : nous allons d'abord détailler les cas d'utilisation identifiés précédemment au moyen de diagrammes d'activité. Nous allons ensuite identifier les données métier qui sont soit manipulées par le système, soit visibles pour son environnement. Ensuite, il faut construire le modèle d'architecture logique au moyen de diagrammes de blocs (de définition et internes) et de diagrammes d'états pour décrire le système comme un assemblage d'éléments communiquant entre eux et s'offrant des services concourant à la réalisation des capacités attendues du système. Enfin, nous terminons cette phase par la description du modèle d'architecture physique pour répondre à la question : quels sont les composants physiques requis pour la fabrication du produit ;

3) la traçabilité et les allocations (détaillés en section 5.4) : la vocation de cette activité est de consolider les différents modèles élaborés au cours du projet. Cette consolidation se fait essentiellement en créant des liens de traçabilité sur trois axes –

Chapitre rédigé par Loïc FEJOZ, Philippe LEBLANC et Agusti CANALS.

liens de satisfaction des éléments de l'architecture logique vers les exigences amont, liens d'allocation des éléments du modèle fonctionnel (les cas d'utilisation) vers les éléments logiques, enfin liens d'allocation des éléments logiques vers les éléments de l'architecture physique ;

4) le modèle de test (détailé en section 5.5) : cette activité conclut les travaux de modélisation. Il s'agit de constituer un référentiel de jeux de test au moyen de diagrammes de séquence et optionnellement de diagrammes d'activité. Ces jeux de test permettront l'acceptation du système à sa livraison.

Dans la pratique, ces activités de modélisation vont être menées dans un projet unique. Ce projet contiendra tous les produits de ces activités, généralement sous la forme de *packages* indépendants. La structure de ce modèle SysML reflète la chronologie des activités :

- *package* « Context » : contexte du système ;
- *package* « System Specification » : spécification du système (hors de son contexte) contenant :
 - *sous-package* « General Needs » : cas d'utilisation et traçabilité avec les exigences,
 - *sous-package* « Technical Needs » : exigences soit directement créées en SysML, soit remodélisées par importation de documents de spécification textuelle, et liens de satisfaction avec les éléments de conception ;
- *package* « System Design » : conception logique et physique du système :
 - *sous-package* « Functional Architecture » : description détaillée des cas d'utilisation,
 - *sous-package* « Domain-specific Data » : description des données du domaine ou du métier,
 - *sous-package* « Logical Architecture » : conception du système en blocs logiques et données métier,
 - *sous-package* « Physical Architecture » : conception du système en blocs physiques,
 - *sous-package* « Allocations » : ensemble des liens de satisfaction entre exigences et éléments logiques, d'allocation entre éléments fonctionnels et éléments logiques et d'allocation entre éléments logiques et éléments physiques ;
- *package* « System Test » : jeux de test pour le système.

Nous rappelons que la chronologie des activités proposée ici ne fait pas partie de SysML qui est neutre par rapport aux processus. Toutefois, elle reflète une communauté de pratiques couramment admises dans l'industrie. Les référentiels [HAS 12, KAP 07, FIO 12] font autorité dans ce domaine.

Bien sûr ce processus est à adapter en fonction de la complexité et de la criticité du système, ainsi que de l'organisation mise en place s'agissant des rôles et compétences

des intervenants et de la répartition géographique des équipes (équipes locales *versus* équipes distribuées). Le guide [SE2 11] fournit de nombreuses idées d'organisation du modèle issues de la mise en pratique sur un modèle de grande taille.

Cette proposition est bien appropriée à la nature de notre étude de cas : un système de complexité moyenne ayant peu d'interactions avec le monde extérieur. En revanche, notre système étant critique, nous allons compléter cette modélisation SysML par d'autres types de modélisation qui seront détaillées aux chapitres 6 et 7.

Ces activités sont en général exécutées en séquence. Néanmoins, il y a de fréquents allers-retours entre les différents modèles afin d'assurer leur cohérence. Cette cohérence est partiellement prise en charge par les outils de modélisation qui sont capables de propager une modification locale à la totalité du modèle, comme le renommage d'un élément (un outil de dessin ne pourrait pas propager automatiquement les modifications).

Faute de place, le travail de modélisation présenté dans ce chapitre n'est pas exhaustif ; il ne s'agit pas ici de réaliser un vrai pacemaker. Nous allons donc porter nos efforts de modélisation sur le générateur de pulsations dans sa phase ambulatoire, c'est-à-dire sa phase utile pour le patient, après implantation et avant explantation. Nous ne détaillerons pas les autres parties du système comme le moniteur de contrôle.

5.2. Spécification du système

Dans cette phase, nous allons décrire le contexte, identifier les cas d'utilisation attendus du système et intégrer dans ce modèle les exigences textuelles afin d'établir la traçabilité entre les exigences et les cas d'utilisation.

5.2.1. Contexte

Le contexte du système doit être décrit par sa structure et par ses phases principales de vie, ce qui est fait au moyen des deux diagrammes « Context » et « Lifecycle ».

Diagramme « Context ». Le contexte (figure 5.1) a comme objectif de présenter le système dans son environnement. Les personnes interagissant avec le système, comme les utilisateurs, opérateurs, etc., sont représentés par des acteurs. Les autres éléments du monde extérieur qui interagissent aussi avec le système sont représentés par des blocs. Le système est représenté aussi par un bloc.

Détails : nous avons créé un bloc racine « Context » pour représenter le domaine complet de notre étude de cas. Ce bloc contient, par des compositions, tous les éléments concernés : le pacemaker représentant le système étudié, ainsi que les personnes et

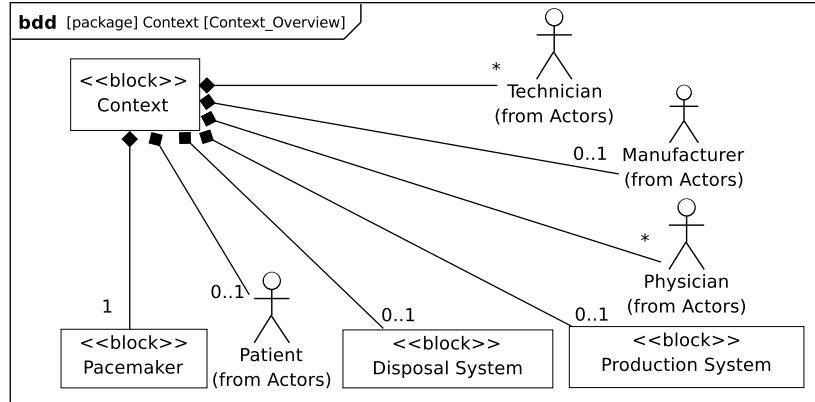


Figure 5.1 – Contexte du système en SysML

systèmes externes interagissant avec lui. Les multiplicités sont précisées du côté des éléments contenus. Le contexte se compose du système « Pacemaker », d’au plus un patient (le patient n’étant pas présent lors de la phase de fabrication du pacemaker), d’un ensemble de médecins et de techniciens (qui ne sont pas constamment présents le long de la vie d’un pacemaker, comme par exemple lors de la fabrication ou du retrait du pacemaker), d’un fabricant (qui disparaît du contexte une fois le pacemaker en exploitation), d’un système de fabrication du pacemaker et d’un système de retrait (eux aussi apparaissant et disparaissant selon la phase dans laquelle est le pacemaker). Deux acteurs différents « Physician » et « Technician » ont été créés pour représenter les deux rôles joués par le personnel médical par rapport au pacemaker : le médecin (« Physician ») fait les interventions chirurgicales, tandis que l’infirmier (« Technician ») ne pratique que des contrôles.

Discussion : nous avons fait le choix de représenter les systèmes externes par des blocs et donc ainsi de les différencier visuellement des personnes qui sont elles représentées sous la forme d’acteurs. Il existe d’autres pratiques qui proposent de les représenter sous la forme d’acteurs, comme pour les personnes, mais marqués d’un stéréotype spécifique comme « external system ».

Diagramme « Lifecycle ». Le cycle de vie du système (figure 5.2) décrit les différentes phases dans lesquelles le système passe au cours de sa vie, de sa fabrication à son retrait. Nous avons utilisé un diagramme d’états attaché au bloc « Context » pour le représenter.

Détails : les sous-phases « Regulation » et « Routine Follow-up » s’exécutent en parallèle dans la phase « Ambulatory ». La spécification fournie ne précise pas si ses traitements sont exclusifs ou non, nous avons considéré qu’il était préférable pour le

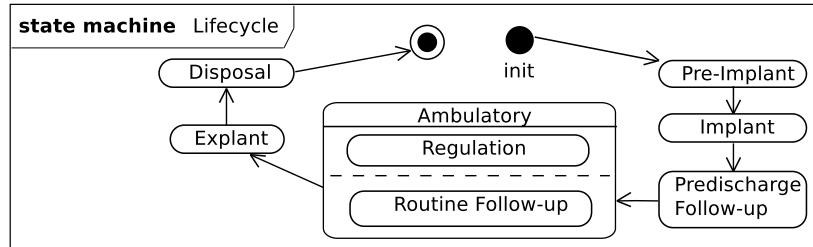


Figure 5.2 – Cycle de vie du système

patient de les faire exécuter en parallèle, ce qui est compatible avec les contraintes techniques exprimées.

Discussion : le cycle de vie décrit ici permet d'expliciter toutes les phases possibles du pacemaker et ainsi, lors de l'élaboration de la meilleure solution, de considérer toutes ces phases en ne se limitant pas seulement à la phase ambulatoire, mais en prenant aussi en compte des phases comme le retrait. La meilleure solution en phase ambulatoire n'est pas forcément la meilleure solution sur le cycle de vie. Nous avons aussi voulu rester à un haut niveau, ainsi nous n'avons pas précisé les conditions de passage d'une phase à l'autre (d'un état à l'autre), ni non plus les actions qui seraient exécutées.

5.2.2. Modèle des besoins et scénarios opérationnels

Nous allons identifier les capacités principales du système en analysant les exigences textuelles exprimées précédemment afin de les représenter sous forme de cas d'utilisation. Comme son nom l'indique, un cas d'utilisation correspond à un ensemble de scénarios similaires concourant au même objectif, à une même utilisation. Des exemples de la vie quotidienne sont : retirer des espèces à un guichet automatique bancaire (que cela se passe bien ou mal), réserver un livre dans une bibliothèque (qu'il soit disponible ou non), etc. Typiquement, un cas d'utilisation va regrouper les scénarios nominaux d'utilisation et les scénarios alternatifs, dégradés ou redoutés pour ce service attendu du système.

Une difficulté fréquemment rencontrée en modélisation par cas d'utilisation est de trouver le bon niveau de granularité d'un cas. Un cas d'utilisation est un service complet offert à l'utilisateur, incluant toutes les variations de ce service. Ce n'est pas une action du système, comme émettre un bip en cas d'anomalie ou imprimer un ticket à la fin d'une transaction bancaire. Un cas d'utilisation doit avoir un objectif plus large, qui justifie l'utilisation du système par un des acteurs en répondant à des questions comme : « pourquoi l'acteur se déplace ? » et « quels bénéfices (résultats visibles) l'acteur obtient-il du système ? ». On estime souvent à une vingtaine maximum le nombre

92 Systèmes embarqués

de cas d'utilisation à traiter pour un projet de modélisation donné. Enfin il est usuel de nommer les cas d'utilisation au moyen de phrases verbales – « faire quelque chose » – et dans le vocabulaire des acteurs impliqués, comme « enregistrer la transaction bancaire » et non dans un vocabulaire technique qui relèverait plutôt d'un choix de conception comme « enregistrer la transaction dans une base de données ».

Nous avons créé un diagramme de cas d'utilisation (figure 5.3), pour identifier l'ensemble des services offerts par le système. Nous avons aussi identifier les services principaux offerts par les systèmes externes présents dans l'environnement (production et retrait). Les cas d'utilisation seront ensuite complétement décrits en phase de conception système.

Nous avons aussi élaboré le diagramme de séquence « ConOps » (figure 5.4) pour montrer le scénario opérationnel global correspondant au cycle de vie nominal d'un pacemaker.

Diagramme « General Needs ». Ce diagramme SysML de cas d'utilisation (figure 5.3) détaille l'usage que les acteurs font des systèmes - acteurs et systèmes ayant été définis dans le contexte (figure 5.1).

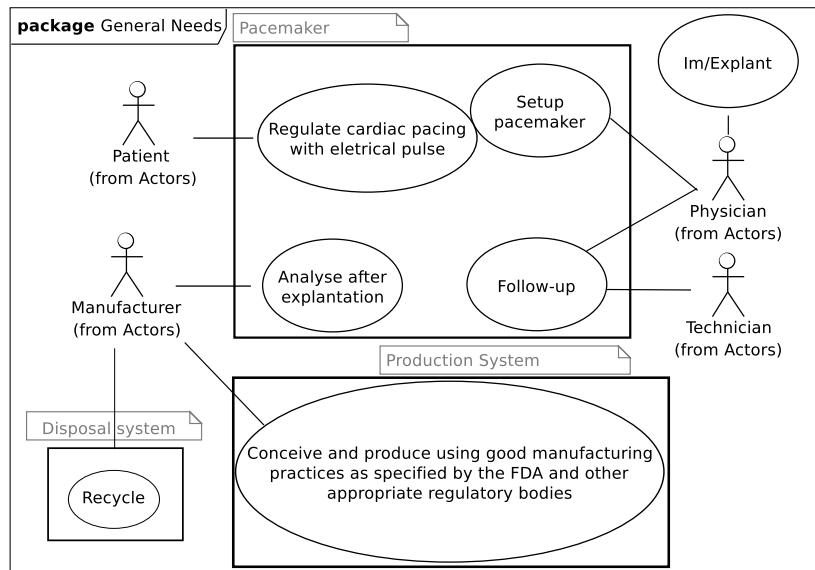


Figure 5.3 – Cas d'utilisation du contexte

Détails : le système « Pacemaker » supporte quatre cas d'utilisation offerts à ses différents acteurs. Nous détaillerons en conception système les trois cas d'utilisation

« Regulate... », « Setup... » et « Follow-up ». Le cas « Analyze after explantation » ne sera pas détaillé car il ne se déroule pas dans la phase ambulatoire du système. Ces cas ainsi que les acteurs impliqués ont été identifiés à partir des exigences textuelles. La traçabilité sera décrite ultérieurement.

Nous avons identifié les capacités principales des systèmes de production et de retrait. Elles ne sont pas détaillées dans la suite du livre, mais elles devraient l'être dans un projet industriel.

Discussion : dans un véritable projet de modélisation, il est fréquent de compléter ce diagramme par des fiches descriptives, une fiche par cas, respectant un format standard du type : « Acteurs », « Préconditions », « Post-conditions », « Scénario nominal », « Exceptions ». Cette documentation est à destination des parties prenantes qui ne sont pas nécessairement familières avec SysML. Nous ne le ferons pas ici car notre objectif est plutôt de faire de la modélisation que de la documentation.

Scénario « ConOps » nominal pour le système. Le diagramme de séquence en figure 5.4 décrit un scénario opérationnel de la vie du système dans son environnement.

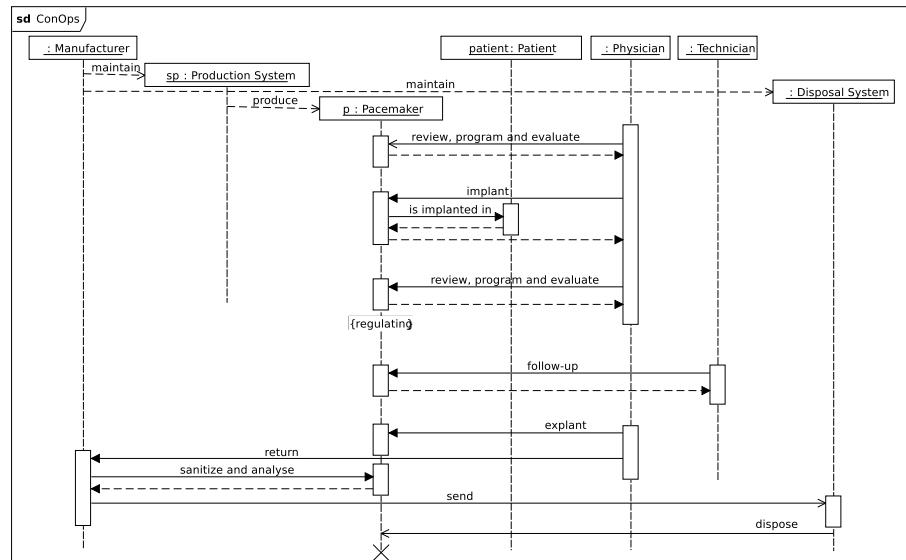


Figure 5.4 – Scénario opérationnel nominal du système dans son contexte

Détails : les lignes de vie correspondent au système « Pacemaker », à ses acteurs et aux autres systèmes interconnectés. Cette séquence montre le déroulement nominal de la vie d'un pacemaker, de sa création, implantation, phase ambulatoire avec suivi,

explantation jusqu'à son retrait. Les interactions « maintain » et « produce » correspondent à la création des trois systèmes : production, retrait et pacemaker. Les interactions suivantes entre les lignes de vie correspondent à des messages synchrones avec des régions d'exécution et des retours. L'instance de système « Pacemaker » disparaît lorsque le pacemaker est envoyé dans le système de retrait. La condition {regulating} ajoutée sur la ligne de vie Pacemaker indique le mode de fonctionnement dans lequel est le système après l'appel de l'opération de « review, program and evaluate » initié en début de la séquence.

Discussion : les interactions ont été représentées par des messages synchrones (appelant bloqué sur le traitement de l'appelé), nous aurions aussi pu les représenter par des messages asynchrones (appelant non bloqué), sans région d'exécution, ni flèches de retour. Mais dans ce contexte, l'usage des messages synchrones est représentatif du mode de fonctionnement attendu car ils imposent la séquentialité des événements. Par exemple, il n'est pas envisageable de faire l'implant avant la revue et l'évaluation.

5.2.3. Modèle des exigences

Deux questions sont toujours à garder en tête lors du développement d'un modèle : « Est-ce un modèle correct d'un point de vue SysML ? » et « Est-ce le bon modèle d'un point de vue fonctionnel ? »

Pour répondre à la première question, il faut d'une part posséder une bonne connaissance de la norme SysML et d'autre part avoir acquis un savoir-faire applicatif. Les outils de modélisation sont aussi là pour aider à construire des modèles SysML corrects du point de vue du langage.

La solution généralement préconisée pour répondre à la seconde question est de tracer le modèle élaboré sur les exigences d'entrée. En effet, comme le fonctionnel est spécifié dans les exigences, prouver que l'on est en train d'élaborer la solution attendue se fera en montrant que les exigences sont satisfaites (avec des liens de traçabilité) par les éléments de modélisation constituant le modèle.

Il faudra montrer que le modèle est suffisant : toutes les exigences sont liées à des éléments du modèle. Pour les systèmes critiques, il faudra aussi montrer que le modèle est nécessaire, c'est-à-dire qu'il n'en fait pas plus qu'attendu : tous les éléments structurants (cas d'utilisation, blocs logiques et blocs physiques) sont liés à des exigences. Afin d'établir cette traçabilité, un travail préliminaire est de rapatrier dans le modèle les exigences textuelles exprimées en phase de spécification. Ces exigences seront généralement importées dans un *package* unique à la racine du modèle. Néanmoins, étant donné le grand nombre possible d'exigences à importer, ce *package* racine peut être découpé en plusieurs *sous-packages*, chaque *sous-package* représentant alors un chapitre ou une section de chapitre du document de spécification des exigences.

Diagramme « Technical Needs : le pacemaker fonctionne correctement ». Ce diagramme SysML d'exigences (figure 5.5) montre des exigences rapatriées du document de spécification qui concernent le bon fonctionnement du pacemaker. Ces exigences ont été liées à des éléments du modèle fonctionnel. Une exigence est modélisée par un nom, un identifiant et un texte. Les exigences peuvent être structurées entre elles au moyen du lien de dépendance « refine »¹ pour exprimer qu'une exigence se décompose en un ensemble d'exigences plus spécifiques.

Nous nous intéressons dans un premier temps aux exigences. Nous parlerons ensuite de la traçabilité avec les cas d'utilisation.

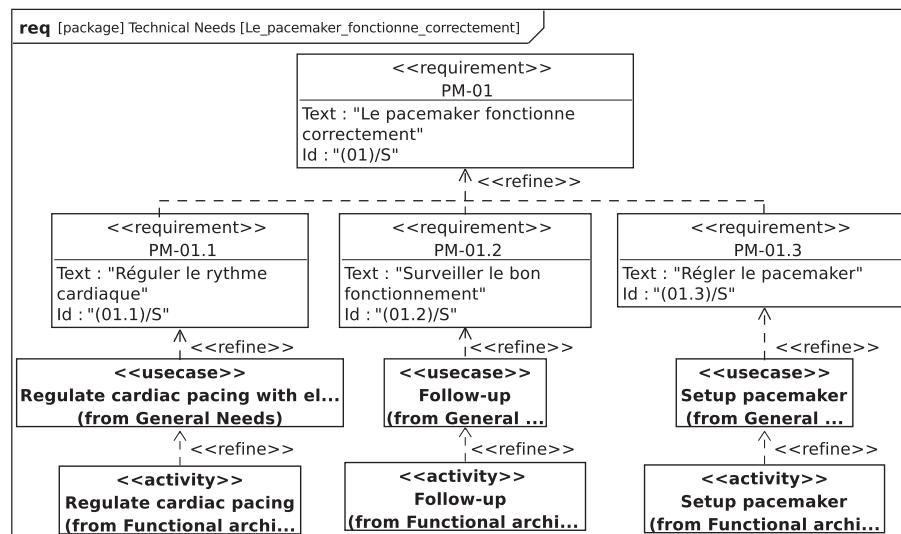


Figure 5.5 – Exigences sur le bon fonctionnement du système

Détails : nous avons décomposé l'exigence PM-01, identifiée (01)/S dans la documentation, en trois exigences PM-01.1, PM-01.2 et PM01-3 au moyen de la relation « refine ». Pour indiquer que cette décomposition en trois sous-exigences est une partition (c'est-à-dire que les trois sous-exigences combinées répondent en totalité à l'exigence supérieure), nous avons utilisé la représentation en forme d'arbre. Si nous avions utilisé des flèches directes entre exigence supérieure et sous-exigences, cela aurait été pour indiquer que la décomposition n'est pas une partition et que l'exigence PM-01 n'est pas entièrement satisfaite par les trois sous-exigences visibles.

1. Les notes de blog [FEJ 10, FEJ 11] justifient l'utilisation de ce type de liens.

Discussion : afin d'organiser hiérarchiquement les exigences, nous avons utilisé le lien de dépendance « refine ». Nous aurions pu utiliser à la place la composition. Ce choix relève des pratiques définies dans l'entreprise, les deux usages étant corrects dans notre étude de cas. L'un des avantages du « refine » est la possibilité d'exprimer des alternatives comme indiqué dans [FEJ 11].

Les concepts d'exigence et de raffinement en SysML ne permettent pas de vérifier les exigences, ni leur partitionnement. Pour le faire, il faut mettre en œuvre des techniques de vérification qui sont abordées dans la suite de ce livre.

Traçabilité Besoins-Cas d'utilisation. Une fois les exigences rapatriées dans le modèle, il faut les relier aux cas d'utilisation identifiés de manière à justifier que l'analyse pratiquée est bien nécessaire et suffisante. C'est ce que l'on peut voir sur la figure 5.5 montrant par des relations « refine » comment les exigences sont prises en compte par les cas d'utilisation.

Détails : le diagramme 5.5 montre aussi, toujours au moyen de relations « refine », par quel diagramme d'activité est détaillé chaque cas d'utilisation, mais cela concerne plutôt la section «Conception Système».

Discussion : nous avons fait le choix d'utiliser la dépendance « refine » entre les cas d'utilisation et les exigences. SysML propose aussi la dépendance « satisfy », mais nous avons préféré la réserver pour exprimer la traçabilité entre exigences et architecture logique.

Dans le cas précis de PM-01, les trois exigences techniques PM-01.X sont prises en compte chacune par un seul cas d'utilisation, mais nous aurions pu avoir des relations M-N : une exigence peut être liée à plusieurs cas d'utilisation et inversement un cas d'utilisation peut être lié à plusieurs exigences.

Afin de démontrer qu'un modèle est nécessaire et suffisant, il faut que toute exigence soit liée à au moins un cas d'utilisation et inversement que tout cas d'utilisation soit lié à au moins une exigence. Comme c'est un travail coûteux en effort, nous verrons cela surtout pour des projets de systèmes critiques.

5.3. Conception du système

En phase de conception du système, nous allons détailler le comportement attendu pour les cas d'utilisation identifiés en phase de spécification, puis proposer une architecture de pacemaker. Les cas d'utilisation sont décrits dans la partie fonctionnelle du modèle, au paragraphe suivant.

La solution est décrite en trois parties :

- données métier : cette partie du modèle définit les données échangées entre les acteurs et les systèmes externes – ce qui correspond aux interfaces du pacemaker – et les données consommées ou produites par le pacemaker ;
- architecture logique : cette partie décrit les constituants logiques proposés pour former le pacemaker ; ils offrent des fonctionnalités qui mises ensemble fournissent les services attendus au niveau du système ; à ce stade, nous ne cherchons pas à indiquer si tel constituant ou telle fonctionnalité est rendu par du logiciel, de l'électronique, de la mécanique, etc.
- architecture physique : cette partie décrit la conception physique qui est proposée comme solution ; nous y identifions les constituants matériels et les constituants logiciels.

La phase de conception se conclue sur des activités de consolidation :

- traçabilité entre exigences et architecture logique : les exigences sont reliées aux constituants logiques pour démontrer l'adéquation de la solution aux besoins ;
- allocation de l'architecture logique sur l'architecture physique : les constituants logiques sont alloués sur les composants physiques pour montrer comment se déploie le système sur l'environnement matériel.

Ces activités de traçabilité et d'allocation seront détaillées dans la section 5.4.

5.3.1. Modèle fonctionnel

En phase de spécification, nous avons identifié quatre cas d'utilisation pour notre système. Trois seulement sont d'intérêt pour nous car ils concernent la partie ambulatoire du pacemaker : « Setup pacemaker », « Regulate cardiac pacing... » et « Follow-up ». Nous détaillons chacun de ces trois cas d'utilisation au moyen d'un diagramme d'activité pour préciser l'enchaînement attendu des actions qui réalisent le cas d'utilisation concerné. Chacun de ces diagrammes a été associé à son cas d'utilisation par un lien « refine » vu en figure 5.5.

Diagramme d'activité de « Setup pacemaker ». Ce diagramme SysML d'activité (figure 5.6) détaille le cas d'utilisation « Setup pacemaker ».

Détails : comme décrit au chapitre 3, le réglage du pacemaker se fait en deux fois, une première fois pendant l'implantation, puis une seconde fois après. Les réglages pendant l'implantation consistent à vérifier l'état de la batterie, puis à programmer le système et régler ses paramètres, enfin à faire des mesures. Les réglages après implantation consistent à refaire des mesures, reprogrammer les paramètres en conséquence, puis sortir rapport et graphiques.

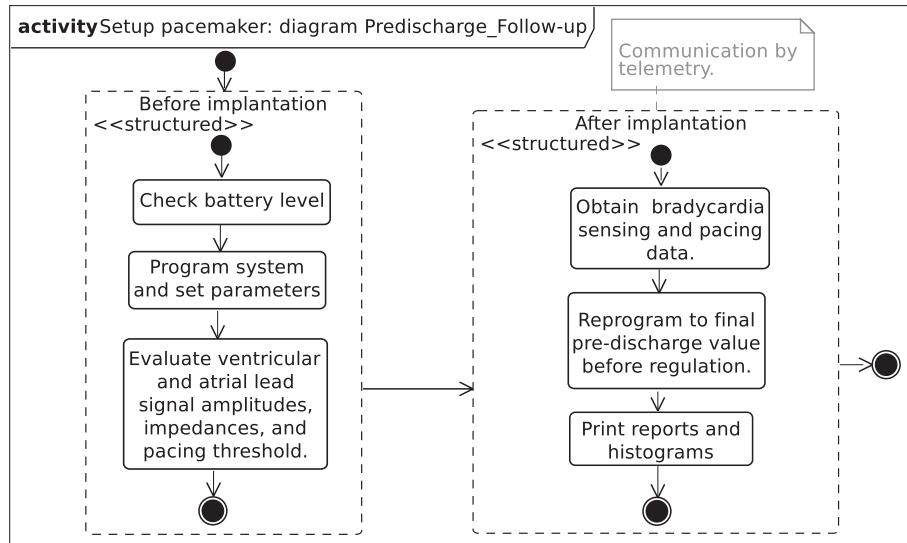


Figure 5.6 – Diagramme d’activité de « Setup pacemaker »

Discussion : les deux grandes étapes sont modélisées par deux blocs d'action et chaque opération à effectuer par le personnel médical est représentée par une action. Nous utilisons des flots de contrôle entre actions pour préciser leur enchaînement. Nous y mentionnons des gardes «[true]» pour signifier qu'une fois l'action terminée, la suivante est automatiquement enclenchée.

Diagramme d'activité de « Regulate cardiac pacing ». Ce diagramme d'activité (figure 5.7) détaille le cas d'utilisation « Regulate cardiac pacing with electrical pulse ».

Détails : le fonctionnement de ce cas d'utilisation se résume à une boucle dans laquelle on réalise en séquence des mesures, un contrôle, un ajustement et enfin l'envoi de l'impulsion, puis le traitement redémarre du début. Le contrôle est fait par rapport au mode et délivre une valeur enregistrée dans un *data-store*. L'action d'ajuster va lire cette valeur dans le *data-store*, puis fournir une valeur ajustée à l'action d'envoyer l'impulsion. En SysML, les données sont véhiculées par des flots de données (*object flows*) connectés aux actions par des ports orientés, tandis que les flots de contrôle (*control flows*) ne font qu'exprimer une séquentialité entre actions. Nous avons ajouté deux commentaires pour préciser la signification ou l'impact de certaines actions.

Diagramme d'activité de « Follow-up ». Ce diagramme SysML d'activité (figure 5.8) détaille le cas d'utilisation « Follow-up ».

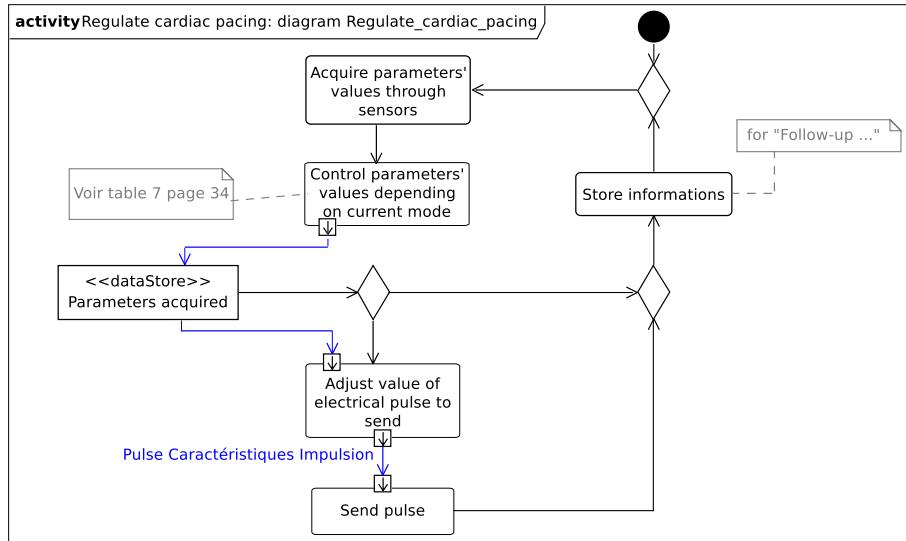


Figure 5.7 – Diagramme d’activité de «Regulate cardiac pacing»

Détails : ce service se déroule en deux étapes, la première permet d’acquérir les mesures par connexion sans fil, la seconde sert à valider les changements s’il y en a eu. Pendant l’étape d’acquisition, deux flots d’actions se déroulent en parallèle : le flot de gauche enchaîne connexion, vérification des états, acquisition, analyse et enfin production de rapports ; le flot de droite concerne la mise à jour des paramètres pour régler le pacemaker. Le parallélisme des flots est représenté par l’usage d’un montage *Fork-Join* (barres verticales). Après l’étape d’acquisition, les éventuels changements de paramètres de fonctionnement doivent être validés. Les histogrammes sont effacés en fin de traitement. Nous avons aussi ajouté quelques commentaires pour clarifier certaines actions.

Discussion : lorsque l’on décrit le comportement d’un cas d’utilisation, la question se pose de la granularité et la précision dans la description des actions. Le niveau de détail doit être cohérent avec l’objectif global qui a été fixé pour le modèle. Si l’on souhaite alors ajouter des détails sur une action, on pourra utiliser des commentaires.

5.3.2. Données métier («Domain-specific Data»)

Cette partie du modèle spécifie les interfaces du pacemaker avec son environnement, ainsi que les données significatives qu’il manipule (produites/consommées). Elles relèvent d’une connaissance du domaine (du métier). Elles sont décrites dans des diagrammes de définition de blocs au moyen des concepts SysML suivants : interface,

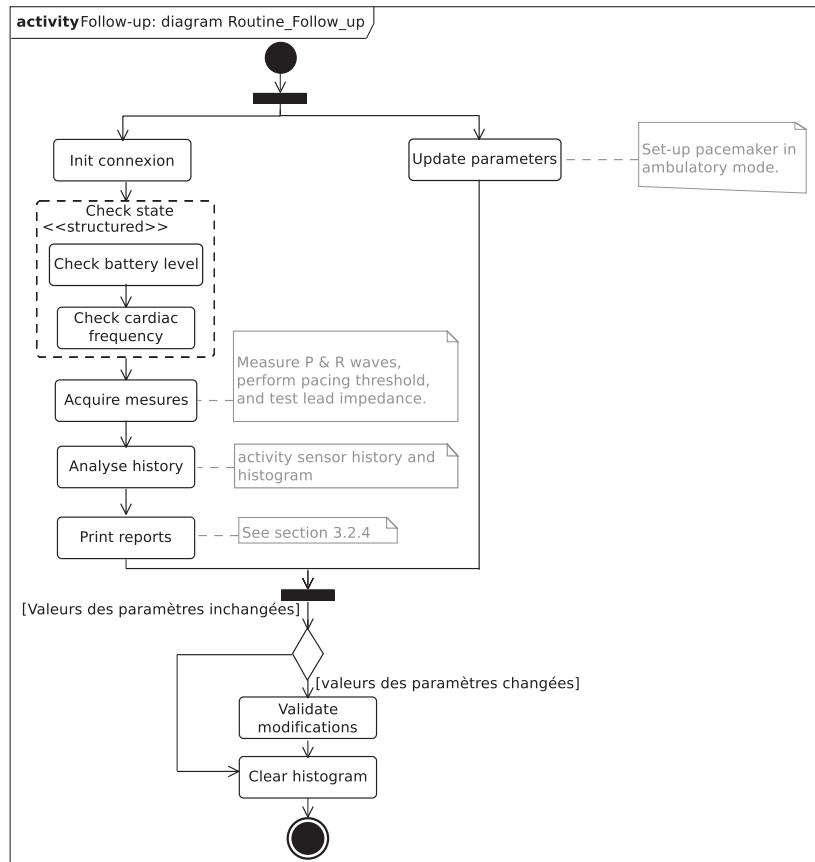


Figure 5.8 – Diagramme d'activité de « Follow-up »

événement, type simple, type complexe représenté par un bloc, type de donnée (*datatype*) et type de valeur (*valuetype*). Nous ne présentons ici que deux diagrammes de ce type, mais le modèle en contient d'autres, comme par exemple la liste des modes supportés par le pacemakers avec leurs types de valeur associés (et leur valeurs : min, max, incrément, valeur nominale, tolérance), les mesures de potentiel, etc.

Diagramme de Données métier « Mesure de fréquence (PPM) ». La figure 5.9 est un diagramme de définition de bloc regroupant un ensemble de déclarations : « unit », « dimension », « valuetype » et « datatype ». Celles-ci servent à préciser les types de données liés aux mesures de fréquence en pulsation par minute (PPM).

Détails : nous avons déclaré « Frequency » comme une « dimension », puis l’unité « ppm » comme une « unit » de dimension « Frequency ». Nous déclarons ensuite des

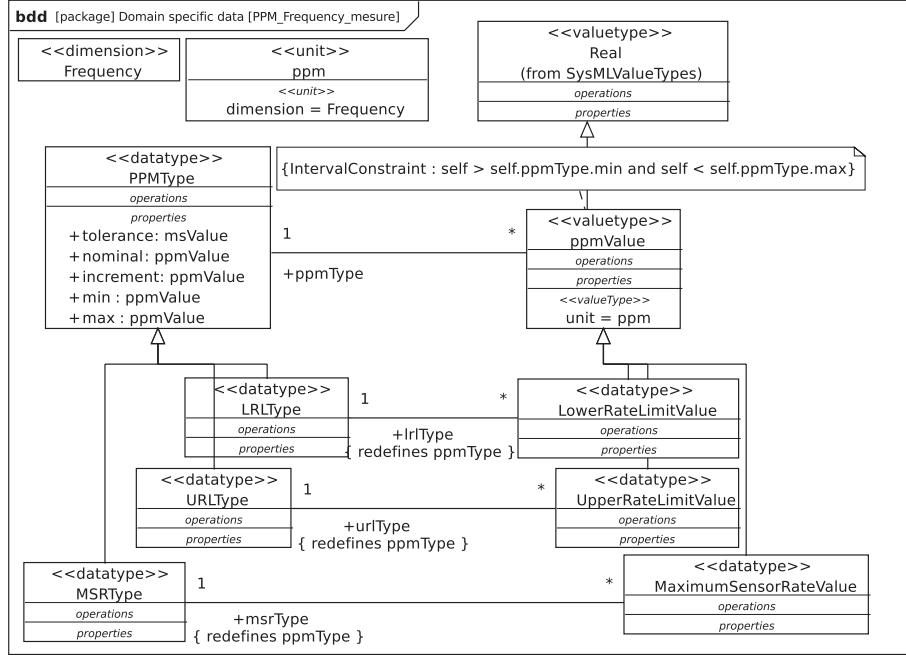


Figure 5.9 – Diagramme de données métier « Mesure de fréquence (PPM) »

« valuetypes » pour représenter des valeurs simples, puis des « datatypes » pour représenter des ensembles de valeurs simples comme « LRLType », « URLType » et « MSRTYPE ». Ces types vont être utilisés pour caractériser les données manipulées par les blocs de l'architecture du système. Nous avons aussi ajouté une contrainte SysML pour préciser l'intervalle des valeurs autorisées (entre min et max). La contrainte « intervalConstraint » exprimée comme un invariant OCL a été ajoutée à la « valuetype » « ppmValue » pour préciser son domaine de validité en fonction des champs « min » et « max » du « PPMType » associé.

Discussion : l'utilisation de « dimension », « unit », « valuetype » et « datatype » est le moyen SysML le plus précis pour déclarer des données. Pour un modèle plus abstrait, nous pourrions préférer l'usage de types de base comme l'entier ou la chaîne de caractères ou de types qui ne seraient pas décrits formellement.

Diagramme de données métier « Mesure de temps ». Ce diagramme SysML de définition de blocs (figure 5.10) sert à préciser les types de données liés aux mesures de temps pour notre pacemaker.

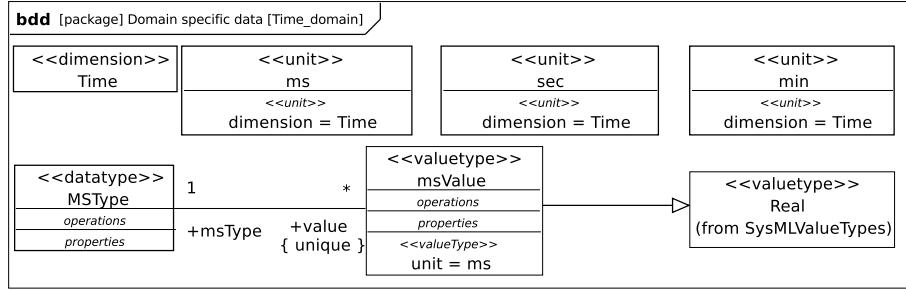


Figure 5.10 – Diagramme de données métier « Mesure de temps »

Détails : nous avons déclaré « Time » comme une « dimension », puis les unités « ms », « sec » et « min » comme des « unit » de dimension « Time ». Nous déclarons ensuite une « valuetype » pour représenter la valeur simple « msValue ». Ce type va nous permettre de construire une donnée complexe comme « MSType ».

5.3.3. Modèle d'architecture logique

Son objectif est de proposer une solution d'architecture. Le système est décomposé en constituants logiques qui chacun offre un ensemble de services. Ces constituants communiquent entre eux pour enchaîner leurs services individuels ce qui permet d'obtenir ainsi au niveau du système les services globaux attendus par son environnement. Cette modélisation se compose de diagrammes de blocs (définition et structure) pour décrire l'architecture et de diagrammes d'états pour décrire le comportement de chacun des éléments logiques.

Diagramme « Logical Architecture ». Ce diagramme de définition de bloc (figure 5.11) décrit la structure du système en constituants logiques.

Détails : le système est décomposé en trois constituants de premier niveau : « Pulse Generator », « Device Controller-Monitor » et « Magnet ». Le bloc « Pulse Generator » se décompose à son tour en trois blocs « PGController », « Battery » et « Lead », ces blocs étant terminaux. Deux contraintes ajoutées dans le bloc « Lead » permet de préciser les impédances tolérées, contraintes qui viennent directement du modèle des exigences.

Les blocs possèdent des ports afin de communiquer des données ou des événements. Prenons l'exemple du bloc « Pulse Generator », il possède :

- deux ports de flots « ElectricalFlow » et « HeartElectricalSignal » pour représenter respectivement le courant électrique produit par la batterie et l'impulsion électrique produite par la « Lead » vers le cœur ;

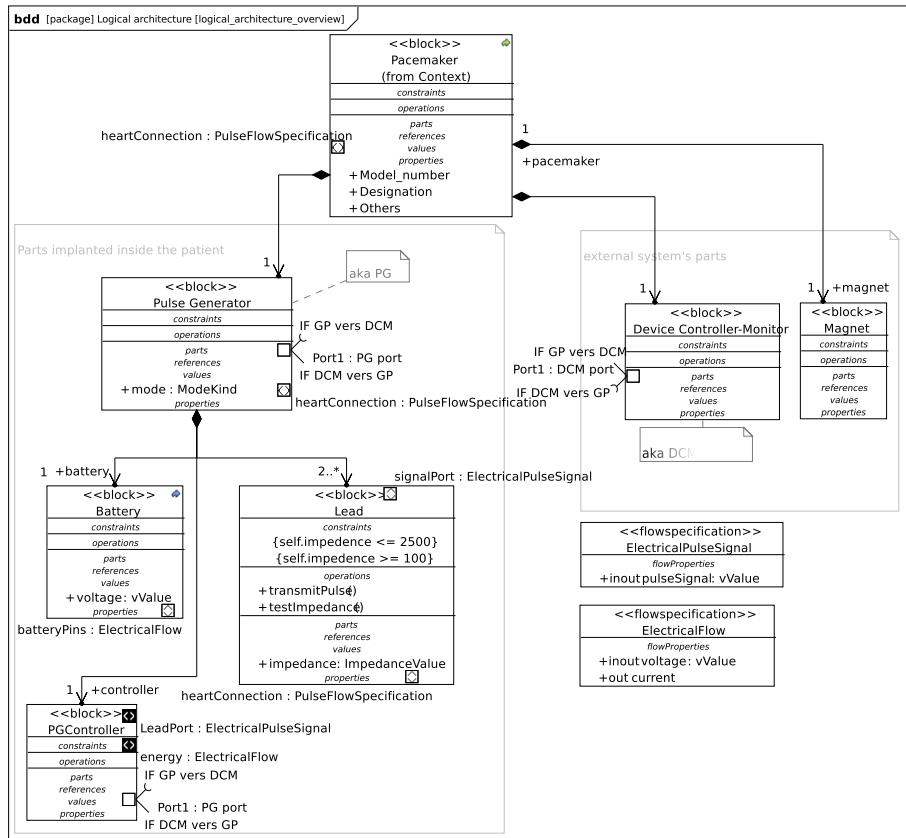


Figure 5.11 – Diagramme d'architecture logique

– un port de contrôle « PG Port » pour recevoir les événements déclarés dans l'interface « IF DCM vers GP » et pour envoyer les événements déclarés dans l'interface « IF GP vers DCM », ce port va permettre de connecter le PG au DCM.

Deux spécifications de flot « ElectricalPulseSignal » et « ElectricalFlow » sont déclarées sur le diagramme pour préciser le contenu des flots véhiculés par les ports des blocs « PGController », « Battery » et « Lead ». Les interfaces déclarées sur les ports de contrôle sont décrites dans un autre diagramme afin de ne pas alourdir le diagramme principal.

Ce diagramme de définition de bloc doit être complété par un diagramme de structure de bloc pour établir la connexion effective entre les ports standards des blocs PG et

DCM. Nous n'avons pas inclus ce diagramme dans ce chapitre, mais il est présent dans le modèle.

De manière à construire une architecture modulaire respectant bien les niveaux de structuration hiérarchique, les ports des constituants les plus bas doivent se retrouver à tous les niveaux en remontant la hiérarchie, évitant en fusionnant, jusqu'au niveau le plus haut, c'est-à-dire le système « Pacemaker ». Un exemple est le port « heartConnection » du constituant « Lead » que nous retrouvons jusqu'au bloc « Pacemaker » : il représente la connexion existante entre le système et le cœur du patient. Les ports des blocs enfants sont connectés aux ports des blocs parents au moyen de diagrammes internes de bloc afin d'assurer la communication entre les différents niveaux de décomposition. La figure 5.12 nous montre un de ces diagrammes internes de bloc. Celui-ci décrit les connexions établies entre le port de contrôle du bloc PG et les ports de ses constituants « PGController », « Battery » et « Lead ».

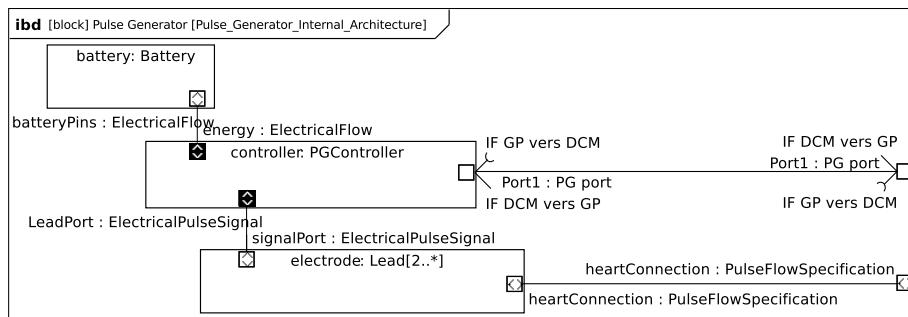


Figure 5.12 – Diagramme interne du bloc « Pulse Generator »

Discussion : afin de faciliter la compréhension de l'architecture logique, dans la figure 5.11, nous avons mis à gauche toute la partie du système implantée dans le patient et à droite les composants qui sont à l'extérieur du corps du patient. La structuration en composants est représentée classiquement par des compositions SysML avec le nom des rôles et leur multiplicité associée.

Les blocs contiennent un certain nombre de propriétés comme *values*, *operations*, *constraints* ou *parts*, qui peut parfois devenir très important. Il faut alors savoir choisir les propriétés que l'on montre dans un diagramme de bloc afin de ne pas le surcharger inutilement. On pourra par exemple utiliser plusieurs diagrammes pour montrer toutes les propriétés des blocs ou faire un diagramme par bloc pour y afficher toutes ses propriétés.

Diagramme d'état du bloc « Pulse Generator ». Ce diagramme d'état SysML (figure 5.13) décrit les évolutions et réactions du bloc PG au cours de son utilisation en fonction des commandes et événements qu'il reçoit.

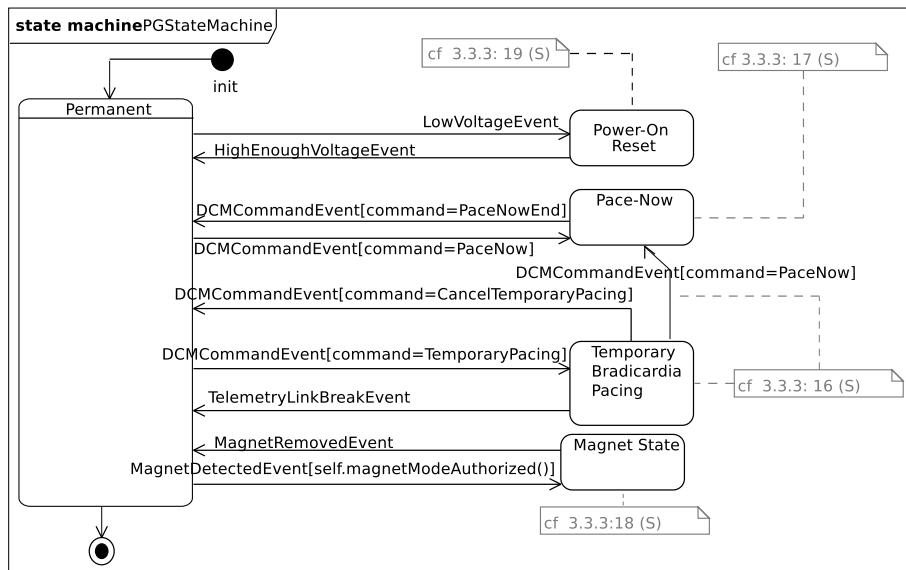


Figure 5.13 – Diagramme d'état du bloc « Pulse Generator »

Détails : le générateur de pulsations possède cinq modes directement identifiés à partir de la spécification des exigences, un mode étant représenté par un état dans le diagramme. Les changements de mode vers ou à partir des états « Pace-Now » et « Temporary Bradicardia Pacing » sont contrôlés par des commandes en provenance du DCM. Les changements de mode vers ou à partir l'état « Power-On Reset » sont contrôlés par des mesures de tension. Les changements de mode vers ou à partir l'état « Magnet State » sont contrôlés par la détection de l'aimant. En général, tous les changements de mode passent par l'état « Permanent » sauf que l'on peut passer directement de l'état « Temporary Bradicardia Pacing » à l'état « Pace-Now » sur réception de la commande « PaceNow ».

Discussion : nous avons ajouté des références vers les différents paragraphes concernés dans la spécification. Une manière plus rigoureuse serait d'établir des relations de traçabilité entre les états de l'automate du PG et les exigences considérées individuellement.

5.3.4. Modèle d'architecture physique

La modélisation architecturale d'un système complexe doit être complétée par un diagramme montrant le déploiement physique du système. Cette description se fait au moyen d'un diagramme de bloc (BDD) SysML contenant des blocs représentant les éléments physiques. On y montre aussi le partitionnement entre matériel et logiciel, ainsi que les moyens de communication employés comme les bus ou protocoles.

Dans la présente étude de cas, un tel diagramme contiendrait des blocs pour représenter la carte électronique supportant le logiciel contrôlant la production des impulsions, les électrodes, la batterie, les fils connectant la batterie aux électrodes, la connexion sans fil au moniteur, etc. Nous ne l'avons pas détaillé ici car il n'apporte pas de concepts SysML nouveaux par rapport à l'architecture logique.

5.4. Traçabilité et allocations

Les activités de traçabilité et d'allocations servent à consolider ensemble les différents modèles élaborés au cours de l'ingénierie du système. La traçabilité entre exigences et architecture va permettre de démontrer que la solution répond aux besoins et que nos choix de conception sont justifiés (nécessaires). La traçabilité va être établie entre les exigences et l'architecture logique. Nous illustrerons l'activité de traçabilité au moyen des deux diagrammes vus en figures 5.14 et 5.15. L'allocation va permettre d'établir des correspondances entre les différentes architectures : comment le fonctionnel est réparti sur le logique et comment le logique se déploie sur le physique. Nous illustrerons l'activité d'allocation au moyen du diagramme vu en 5.16.

Diagramme de traçabilité « Technical Needs : Divers ». Ce diagramme d'exigences (figure 5.14) montre les relations de traçabilité entre des exigences et des éléments d'architecture système, ici des blocs et des attributs de bloc.

Détails : l'exigence de premier niveau PM-03 est d'abord raffinée en trois exigences, PM-03.X. L'exigence de haut niveau PM-03 est satisfaite par le bloc « Pacemaker ». L'attribut « id » du bloc « Pacemaker » satisfait les trois exigences de deuxième niveau PM-03.X. L'exigence PM-03.1 s'appuie sur les attributs « Model_number » et « Designation » du bloc « Pacemaker ». L'exigence PM-04 n'est pas raffinée et est satisfaite par le bloc « Lead ».

Discussion : une note « problem » a été ajoutée sur la décomposition « refine » entre exigences pour indiquer un manque dans la spécification. Elle aurait aussi pu être associée à la relation de traçabilité entre PM-0.3 et « Pacemaker ».

Le lien « satisfy » ne fait qu'indiquer l'intention que le bloc source du lien doit satisfaire l'exigence cible. C'est l'un des objectifs de l'étape de vérification d'en assurer la véracité.

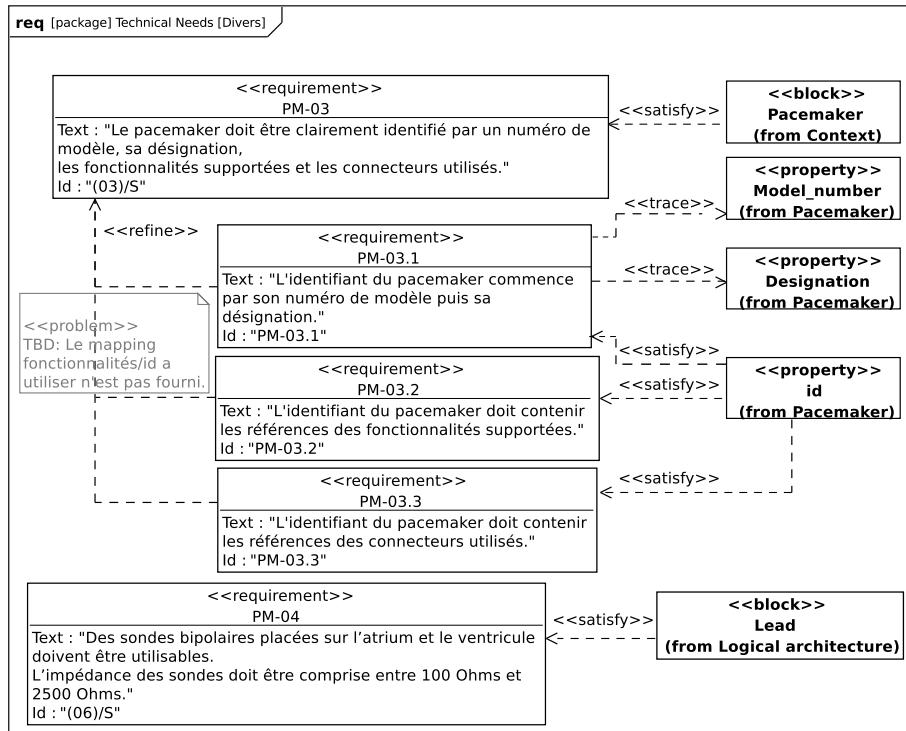


Figure 5.14 – Diagramme de traçabilité entre exigences et blocs et données logiques

Diagramme de traçabilité « Technical Needs : comportement du pacemaker ». La figure 5.15 est un diagramme d'exigences montrant les relations de traçabilité entre des exigences et le comportement défini pour le bloc PG.

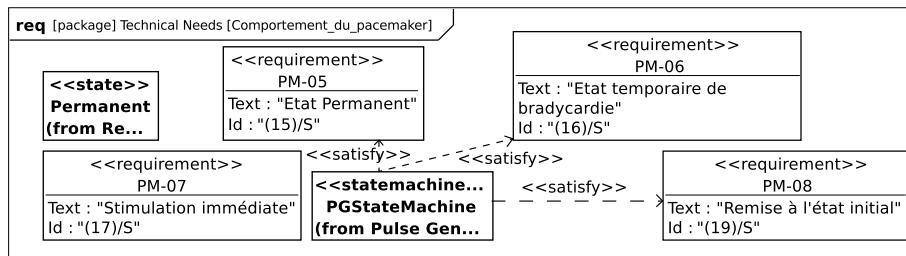


Figure 5.15 – Diagramme de traçabilité entre exigences et comportement

Détails : les différents modes de fonctionnement décrits dans la spécification des besoins sont implémentés par les états de la machine d'état du bloc « Pulse Generator » (figure 5.13).

Discussion : l'exigence PM-07 peut être interprétée comme la possibilité de passer de n'importe quel état du PG à l'état « Pace-Now » (existence d'une transition). Cette exigence est vérifiable sur le modèle, elle s'exprimerait par une contrainte OCL spécifiant qu'il existe bien une transition partant de tout état vers « Pace-Now ». Par exemple, le tutoriel [GAB 09] explique cela dans l'outil Topcased.

Diagramme d'allocation. La figure 5.16 est un diagramme de bloc montrant les relations d'allocation entre les activités (fonctions) identifiées en analyse fonctionnelle et les blocs issus de l'architecture logique.

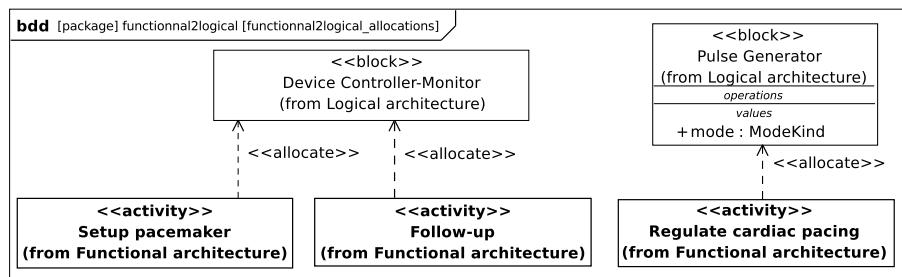


Figure 5.16 – Allocation entre éléments fonctionnels et éléments logiques

Détails : la fonction « Regulate cardiac pacing » (activité SysML) est allouée au bloc « Pulse Generator » tandis que les fonctions « Setup pacemaker » et « Follow-up » sont allouées au bloc « Device Controller Monitor ».

Discussion : ces diagrammes nous permettent d'allouer les fonctions (représentées par des activités dans le modèle fonctionnel SysML) aux blocs de l'architecture logique. Une allocation entre fonction et bloc conduira à créer, suivant les cas, une ou plusieurs opérations dans le bloc cible afin d'implémenter la fonction. Au final, toutes les fonctions devront avoir été allouées sur des blocs, ce contrôle de complétude peut être effectué par l'outil de modélisation SysML.

5.5. Modèle de test

Le modèle de test sert à définir les critères d'acceptation du système. Généralement il est constitué de diagrammes de séquence qui décrivent des scénarios opérationnels détaillés au niveau du système vu comme une boîte noire : les interactions reçues par

la ligne de vie du système correspondent à des entrées ou commandes à appliquer au système sous test, alors que les interactions en provenance du système correspondent à des observations qui doivent être faites par l'opérateur de test. Des diagrammes d'activité peuvent aussi être utilisés, en complément ou en alternative, pour décrire les actions qu'un opérateur de test doit faire pour tester et valider le système.

Une bonne pratique est aussi de décrire en quoi les tests à pratiquer valident le système. Cela est fait en spécifiant des relations de traçabilité entre les jeux de test et les exigences.

Diagramme de traçabilité « System Test : Requirements Verification ». Le diagramme d'exigences vu en figure 5.17 montre quelles exigences sont testées par les jeux de test. Une exigence peut être vérifiée par plusieurs jeux de test et inversement un jeu de test peut vérifier plusieurs exigences.

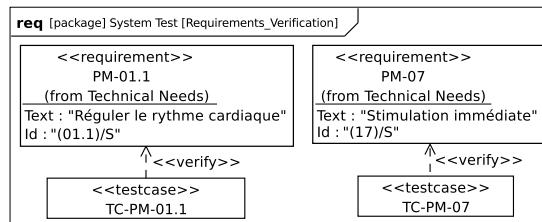


Figure 5.17 – Traçabilité exigences-jeux de test

Détails : le jeu de test TC-PM-01.1 teste l'exigence PM-01.1, et le jeu de test TC-PM-07 teste l'exigence PM-07.

Diagramme de séquence pour le jeu de test TC-PM-07. Le diagramme de séquence de la figure 5.18 montre les interactions à entrer et à observer pour vérifier le test TC-PM-07.

Détails : le test s'exécute entre le système « Pacemaker » et le patient (qui est un acteur). Il consiste en une boucle générale permettant d'exécuter le test autant de fois que voulu pour éprouver le système. Cette boucle contient à son tour deux boucles, l'une s'exécutant dans le mode « Permanent », l'autre dans le mode « Pacenow ».

Discussion : ces diagrammes nous permettent de spécifier les tests dont nous aurons besoin lors de la validation du système. Cela est intéressant à deux titres. D'abord, ils nous forcent à imaginer comment traiter les points durs, aussi bien fonctionnels que techniques – performances, robustesse, etc. Ensuite, ils nous permettent de créer le plan de validation, soit de manière automatique (si l'outil le permet), soit manuellement.

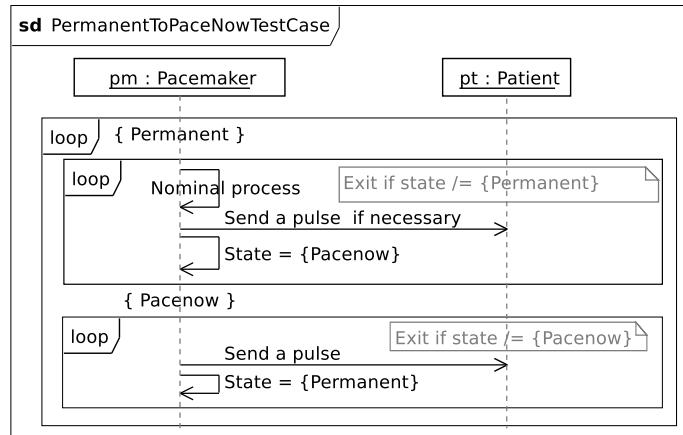


Figure 5.18 – Diagramme de séquence pour le jeu de test TC-PM-07

Bien sûr si l'on souhaite l'exhaustivité, le nombre de tests sera probablement très (trop) important et difficile à gérer. On doit donc trouver le bon compromis entre les efforts investis et la couverture de test recherchée, et se limiter aux tests essentiels. Ceci reste un point dur dans un projet de modélisation SysML et peut être traité avec l'aide d'outils comme des analyseurs de graphe et des vérificateurs de propriétés.

Diagrammes présentant une vue générale des exigences. Les figures 5.19 et 5.20 sont des diagrammes d'exigences montrant les liens entre une exigence et des éléments du modèle.

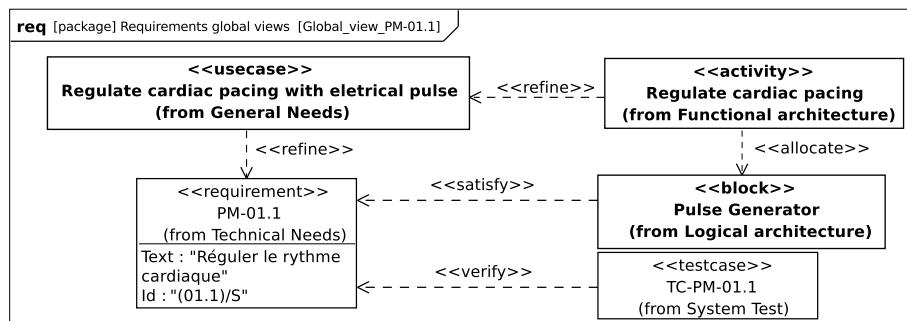


Figure 5.19 – Diagramme présentant l'exigence TC-PM-01.1

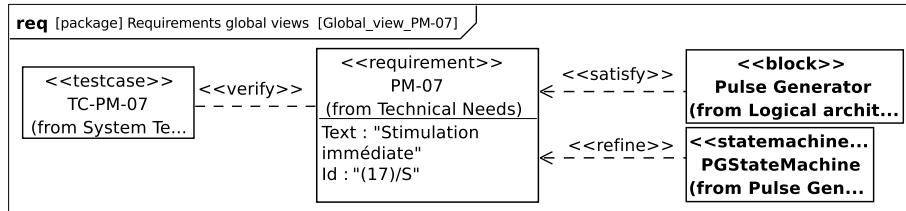


Figure 5.20 – Diagramme présentant l'exigence TC-PM-07

Détails : PM-01.1 est satisfaite par le bloc « Pulse Generator » (PG). Elle est raffinée par le cas d'utilisation « Regulate... », lui même raffiné par l'activité « Regulate... » qui est allouée au bloc « PG ». Enfin elle est vérifiée par le jeu de test « TC-PM-01.1 ».

PM-07 est satisfaite par le bloc « PG ». Elle est raffinée par le diagramme d'état de « PG ». Et elle est vérifiée par le jeu de test « TC-PM-07 ».

Discussion : ces diagrammes sont optionnels. En effet, si nous souhaitons être exhaustifs, il faudrait en faire autant qu'il y a d'exigences, ce qui peut en faire des centaines ! Cependant, leur intérêt est évident, ils permettent de représenter sur un même diagramme une exigence dans son écosystème (relations entretenues avec les parties fonctionnelle et logique de la modélisation), facilitant ainsi la navigation à l'intérieur du modèle complet. On se limitera donc à traiter les exigences majeures. Il est à noter que ces diagrammes n'apportent pas de nouvelles informations, mais ne consistent qu'à représenter en un seul endroit des informations définies dans plusieurs diagrammes. C'est l'apport de la modélisation par rapport à une collection de dessins. Certains outils sont d'ailleurs capables de reconstituer ce type de diagramme automatiquement par analyse du modèle complet.

5.6. Conclusion

L'ingénierie du système Pacemaker guidée par la modélisation nous a conduit à élaborer un modèle global combinant différents aspects de l'ingénierie :

- exigences identifiées à partir du document de spécification ;
- analyse fonctionnelle pour identifier les cas d'utilisation attendus ;
- architecture fonctionnelle pour détailler les cas d'utilisation ;
- architecture logique pour décrire la solution logique à base de blocs communicants ;
- architecture physique pour montrer le déploiement de la solution sur des composants physiques ;
- plan de test pour la validation du système.

Des activités de consolidation sont menées pour rendre toutes ces parties de la modélisation cohérentes entre elles, c'est-à-dire :

- la traçabilité entre les différents niveaux : exigences, fonctionnel et logique ;
- l'allocation entre les différentes architectures fonctionnelle, logique et physique ;
- la traçabilité entre tests d'acceptation et exigences.

L'ensemble de ces modèles décrit complètement et avec précision la solution proposée face aux besoins à couvrir. Cette modélisation constitue la référence pour les étapes suivantes d'ingénierie de la solution. Elle présente aussi comme avantage de montrer la solution avec des niveaux de détails et de formalisation différents, du plus simple avec les cas d'utilisation au plus compliqué avec les ports, interfaces et automates. Cela facilite les échanges et la compréhension par toutes les parties prenantes, du client au fournisseur, sans toujours nécessiter une grande expertise en SysML.

La modélisation apporte aussi deux bénéfices majeurs : la maintenabilité et l'évolutivité de la solution. Il est en effet relativement aisés de la faire évoluer au gré des évolutions. Dans le cas par exemple d'une modification d'exigence, grâce aux liens de traçabilité, nous savons rapidement et précisément quels éléments fonctionnels, logiques ou physiques sont impactés. Dans le cas d'un changement d'interface avec l'environnement, l'architecture logique nous permet d'identifier les blocs impactés. Il est aussi facile de mettre à jour le plan de test en reconSIDérant les diagrammes de séquence par rapport aux modifications faites dans l'architecture logique : les interactions entre lignes de vie doivent s'appuyer sur les opérations et les ports des blocs.

SysML permet enfin de réaliser des analyses de risque, de performance, de coût, de choix d'architecture, de sûreté, etc., en utilisant les « View », « ViewPoint » ou diagrammes paramétriques. A noter que ces derniers peuvent être exécutés via des outils dédiés de type solveurs mathématiques.

5.7. Bibliographie

- [FEJ 10] FEJOZ L., « How to link SysML requirements ? », 2010.
- [FEJ 11] FEJOZ L., « Solution space exploration with SysML requirements », 2011.
- [FIO 12] FIORÈSE S., MEINADIER J., *Découvrir et comprendre l'ingénierie système*, AFIS, Editions Cépadès, 2012.
- [GAB 09] GABEL S., Topcased OCL Tooling Tutorial, 2009.
- [HAS 12] HASKINS C., SE HANDBOOK WORKING GROUP, *INCOSE Systems Engineering Handbook : Version 3.2.2*, International Council on Systems Engineering, 2012.
- [KAP 07] KAPURCH S., *NASA Systems Engineering Handbook*, DIANE Publishing Company, 2007.
- [SE2 11] SE2 CHALLENGE TEAM, Cookbook for MBSE with SysML, 2011.

Chapitre 6

Analyse des exigences

6.1. Introduction

En amont du cycle de vie d'un système complexe, le recueil des exigences joue un rôle déterminant pour les phases d'analyse, de conception, de réalisation et de maintenance qui lui font suite. La traçabilité des exigences au long du cycle de vie est ainsi devenue une préoccupation majeure des praticiens du domaine de l'ingénierie système. Ces mêmes praticiens reconnaissent l'importance d'une analyse d'exigences menée au plus tôt dans le cycle de vie d'un système.

Outre les relectures croisées qui permettent d'apprécier le degré de cohérence des exigences entre elles, il est un type d'analyse essentiel pour détecter au plus tôt les erreurs de conception et réduire le coût de développement et de test du système : la confrontation d'un modèle de conception aux exigences. En contexte SysML, ce type de confrontation met en balance une architecture d'instances de blocs et les comportements des entités qui la composent, avec les exigences exprimées dans les diagrammes d'exigences et les cas d'utilisation.

Pour confronter une conception à des exigences, nous avons besoin de diagrammes d'architecture et de comportements à la fois exécutables et non ambigus. C'est une des clés de l'approche de simulation et vérification formelle mise en œuvre dans ce chapitre.

La levée de toute ambiguïté d'interprétation des diagrammes de conception (instance de blocs et machines à états) d'une part et la traduction de ces diagrammes dans un

Chapitre rédigé par Ludovic APVRILLE et Pierre DE SAQUI-SANNES.

formalisme déjà outillé en matière de vérification formelle d'autre part, demande de doter ces diagrammes de conception SysML d'une sémantique formelle. C'est chose faite avec le langage AVATAR [APV 11a] que nous utilisons dans ce chapitre. AVATAR est le langage supporté par l'outil TTool [APV 11b]. Logiciel libre, TTool est interfacé avec l'outil UPPAAL [BEN 04] en vue de vérifier la logique et la temporalité de conceptions modélisées en SysML.

Ce chapitre réutilise le modèle SysML de pacemaker du chapitre précédent et l'étend en vue d'une analyse logique et temporelle. Nous étendons le diagramme d'exigences SysML original et lions formellement les exigences aux propriétés à vérifier.

Le langage TEPE d'expression de propriétés et son intégration à la méthode orientée « vérification de modèle » supportée par l'outil TTool, sont au cœur de la présentation de l'environnement TTool/AVATAR décrit par la section 6.2. Ce modèle AVATAR du pacemaker comporte une partie « documentaire » formée des diagrammes d'exigences (section 6.3) et d'analyse et une partie rendue « exécutable » que forment les diagrammes de conception d'architecture (section 6.4) et comportement (section 6.5).

La mise au point de ces diagrammes de conception fait appel au simulateur intégré à l'outil TTool. Cette fonctionnalité de débogage est présentée à la section 6.5, et illustrée à l'aide des diagrammes les plus significatifs du modèle AVATAR du pacemaker.

La section 6.6 enchaîne sur la vérification formelle. Parmi les modes de fonctionnements du pacemaker, ce chapitre retient le mode VVI pour la concision des modèles et des traces de simulation. La vérification formelle du même modèle illustre l'intérêt de l'approche « presse bouton » mise en œuvre par TTool sans occulter le problème de la remontée d'informations d'UPPAAL vers TTool. Les diagrammes d'expressions de propriété (langage TEPE) permettent de décrire des propriétés et de les traduire en observateurs utiles pour guider la vérification formelle. L'ensemble forme un environnement de simulation et vérification formelle de modèles SysML temporisés que la section 6.7 compare à d'autres outils et approches SysML.

La section 6.8 synthétise les contributions du chapitre et ouvre des perspectives sur la transition entre le modèle SysML/AVATAR et le modèle AADL du même pacemaker.

6.2. Le langage AVATAR et l'outil TTool

L'approche AVATAR comporte une méthode (section 6.2.1), un langage basé SysML (section 6.2.2), enrichi par TEPE (section 6.2.3) et supporté par l'outil TTool (section 6.2.4).

6.2.1. Méthode

Le langage SysML normalisé à l'OMG est une notation et en aucun cas une méthode. A l'origine, le langage AVATAR se trouvait dans la même situation. Nous lui avons adjoint une méthode applicable à une large gamme de systèmes. Les sept étapes s'enchaînent dans cet ordre avec des rebouclages possibles :

- 1) recueil des exigences ;
- 2) expression des hypothèses simplificatrices ;
- 3) analyse guidée par les cas d'utilisation documentés par des scénarios et des enchaînements d'activités ;
- 4) conception en termes d'architecture et de comportements ;
- 5) expression des propriétés à vérifier ;
- 6) confrontation des diagrammes de conception aux exigences et propriétés par combinaison de techniques de simulations et de vérifications formelles ;
- 7) expression des allocations entre éléments de modèles et construction des matrices de traçabilité.

Un rebouclage peut être effectué entre n'importe quel étape i vers une étape précédente j avec $j < i$, même si l'on préférera toujours aller le plus loin possible – c'est-à-dire jusqu'à la dernière étape 7 – dans chaque itération de la méthodologie.

6.2.2. Diagrammes AVATAR et norme SysML

Le langage AVATAR réutilise les neuf diagrammes SysML, leur syntaxe et leur sémantique dans la mesure où celle-ci est définie de manière suffisamment précise dans la norme [OMG 11]. Le langage AVATAR est de plus outillé et associé à une méthode qui répond aux besoins d'une large classe de système temps réel. Cette méthode oriente la manière dont nous allons maintenant décrire les diagrammes AVATAR et leur positionnement par rapport à la version normalisée de SysML.

En amont de la méthode, le recueil des exigences organise ces dernières dans une structure arborescente qui montre leurs attributs, leurs interrelations et leurs liens avec d'autres éléments du modèle, y compris – en AVATAR – des références aux propriétés à vérifier formellement sur les diagrammes d'architecture fonctionnelle ou organique. Outre certains attributs – de sécurité par exemple – associés à chaque exigence, AVATAR se démarque de SysML par la possibilité de référencer des propriétés qui ont vocation à être ultérieurement décrites de façon plus précise et formelle avec des éléments de la conception.

La phase d'analyse s'appuie classiquement sur un diagramme de cas d'utilisation documenté par des diagrammes de séquences et des diagrammes d'activité. AVATAR

réutilise le diagramme IOD (*Interaction Overview Diagram*) d'UML 2 pour structurer les scénarios que décrivent les différents diagrammes de séquences. Ces derniers supportent non seulement les opérateurs de temps relatif et absolu mais aussi la manipulation de temporisateurs au travers d'opération d'armement, test d'expiration et réinitialisation. En plus de gérer le temps, il faut représenter le caractère réactif des systèmes temps réel. Pour cela, les diagrammes de séquences AVATAR modélisent des interactions asynchrones aussi bien que synchrones. Notons cependant que les flux continus ne sont pas supportés. Cette limitation tient au fait que nous ne disposons pas à ce jour d'outils de simulation et de vérification formelle aptes à traiter des flux discrets et continus au sein d'un même modèle AVATAR.

Cette restriction se retrouve dans les diagrammes de conception qui définissent les entités du système tant en termes d'architecture que de comportement. En termes d'architecture, AVATAR utilise un seul et même diagramme pour représenter d'une part les blocs (donc le typage des éléments d'architecture) et d'autre part, les instances de blocs (qui sont sur le même diagramme composées au sens de la composition entre machines à états temporisées communicantes). Au niveau des comportements, les diagrammes machines à états supportent les intervalles temporels (clause 'after[min, max]') et les opérations sur les temporisateurs déjà énumérées pour les diagrammes de séquences. AVATAR fait ici l'hypothèse d'un système « fermé », c'est-à-dire que les échanges de signaux avec l'environnement du système doivent être explicitement modélisés dans la conception.

Cet ensemble de diagrammes sert de support à la méthode présentée au paragraphe précédent, avec le souci de pouvoir valider l'architecture de conception et les automates de comportements. Pour cela, les diagrammes d'instances de blocs et de machines à états sont dotés d'une sémantique formelle permettant leur traduction vers des automates temporisés que supportent l'outil UPPAAL. Ceci ajoute de la formalisation à SysML.

6.2.3. Le langage TEPE d'expression de propriétés

La formalisation du langage AVATAR et la passerelle établie entre les outils TTool et UPPAAL, participent de la mise en œuvre d'une approche de vérification formelle. Cette approche fait classiquement référence à la notion de « propriété à vérifier ». Dérivées le plus souvent des exigences, ces propriétés peuvent s'exprimer dans un langage formel étranger à SysML. Cette approche où les propriétés se retrouvent *de facto* exprimées dans un formalisme autre que SysML est courante [FAR 06, LEB 10, SAT 10]. *A contrario*, l'introduction des propriétés dans le modèle SysML du système est une spécificité d'AVATAR qui étend ainsi les diagrammes paramétriques de SysML afin de définir un langage graphique d'expression de propriétés : TEPE (*Temporal Property Expression languagE*) [Kno 11a].

Le langage SysML utilise les diagrammes paramétriques pour établir des relations entre attributs des blocs de l'architecture et plus généralement pour définir des équations impliquant des éléments d'un ou plusieurs diagrammes d'un modèle SysML. Ainsi, nous pensons que ces diagrammes paramétriques offrent un excellent cadre structurant pour exprimer des propriétés à vérifier. Avec TEPE, les propriétés sont décrites d'une part au travers de relations logiques et temporelles entre occurrences d'événements et d'autre part à l'aide de relations entre valeurs des attributs de blocs. De ces deux types de relations est né le langage TEPE d'expression de propriétés temporelles qui gagne en facilité d'utilisation par rapport aux formules de logiques, sans pour autant perdre en pouvoir d'expression. La sémantique formelle de TEPE est par ailleurs basée sur les logiques temporelles *Metric Temporal Logic* (MTL) [KOY 92] et *Fluent Linear Temporal Logic* (FLTL) [LET 05].

6.2.4. TTool

Le langage AVATAR est intégralement supporté par l'outil libre TTool [APV 11b] développé pour Linux, Windows ou MacOS. L'installation par défaut de TTool permet de disposer des éditeurs de diagrammes et du simulateur. TTool implante des passerelles vers trois outils développés par d'autres laboratoires : UPPAAL pour la vérification formelle de propriétés logiques et temporelles, Proverif pour la vérification de propriétés de sécurité [ABA 02] et SocLib pour le prototypage virtuel du logiciel et du matériel de systèmes temps réels. L'absence de communication cryptée dans le système du pacemaker et la délégation des implantations logicielles aux approches MARTE et AADL font que nous n'utiliserons ni Proverif ni SocLib dans la suite de ce chapitre. *A contrario*, UPPAAL est utilisé dans ce chapitre pour la vérification de contraintes de sûreté de fonctionnement.

6.3. Expression AVATAR du modèle SysML de pacemaker enrichi

La modélisation en AVATAR est présentée en deux étapes : le fonctionnement et les hypothèses du système modélisé, et la représentation des exigences dudit système.

6.3.1. Fonctionnement du Pacemaker et hypothèses de modélisation

Palliant aux insuffisances d'un cœur qui ne bat plus correctement, un pacemaker en fonctionnement peut se trouver dans un des modes suivants caractérisé par trois lettres :

- la première lettre (A, V ou D) indique qui de l'oreillette, du ventricule ou des deux est stimulé électriquement (*paced* en anglais) et doit recevoir une impulsion ;
- la deuxième lettre (A, V ou D) indique quelle chambre (éventuellement les deux) sera monitorée électriquement et donc observée ;

– la troisième lettre I, T ou D identifie respectivement les modes *Inhibited*, *self Triggering* et *dual Pacing*. Dans le mode I, une contraction cardiaque détectée comme un événement provenant du cœur dans le délai imparti va empêcher le pacemaker d'envoyer une impulsion électrique. Dans le mode T, un événement provenant du cœur enclenchera immédiatement une stimulation de la part du pacemaker. Ainsi, dans le mode VVI, le ventricule est stimulé si un événement ne provient pas du ventricule. Si un événement provient du ventricule alors la stimulation est inhibée. Ce mode particulier sera utilisé dans ce chapitre pour illustrer la vérification formelle guidée par des observateurs.

Ce chapitre ajoute d'autres hypothèses simplificatrices à l'ensemble du modèle. Nous supposerons en particulier que le pacemaker a été correctement implanté et que le patient est en mode déambulatoire. Le pacemaker est réputé correctement initialisé. Ses composants logiciels et matériels ne tombent jamais en panne. Ni sa maintenance, ni son remplacement, ni sa valorisation en tant que déchet ne sont traités. De cette architecture globale, nous extrayons une architecture simplifiée, support à une présentation pédagogique des modes de simulation et de vérification formelle supportés par TTool.

6.3.2. Diagramme d'exigences

Le diagramme d'exigences AVATAR de la figure 6.1 reprend les diagramme d'exigences SysML du chapitre précédent. Il met en évidence à la fois les buts principaux de l'application, les hypothèses de modélisation, les relations de raffinement entre exigences, ainsi que les relations de composition. Nous avons enrichi ce diagramme avec des liens entre exigences et « propriétés à vérifier ». Pour l'instant, ces propriétés sont identifiées par un nom. Elles seront détaillées lorsque les éléments d'architectures tels que les noms de signaux et les attributs de blocs auront été définis.

Notons que pour les besoins de l'exposition dans ce chapitre, nous nous limitons à des propriétés liées au mode VVI. Plus précisément, la verticale gauche de la figure 6.1 est consacrée aux exigences en partant de l'exigence la plus fondamentale (tout problème cardiaque est sous contrôle du pacemaker) vers les plus concrets. La verticale de droite complète le diagramme par des hypothèses sur le patient et le pacemaker, notamment sur le fonctionnement du système lui-même. Au bas de la figure, les relations « verify » lient le nœud d'exigence VVI stimulation aux propriétés que l'on vérifiera à la section 6.6.

6.4. Architecture

La clé de voûte d'une conception AVATAR est le diagramme d'instances de blocs, reflet de l'architecture du système modélisé. La simulation et la vérification formelle

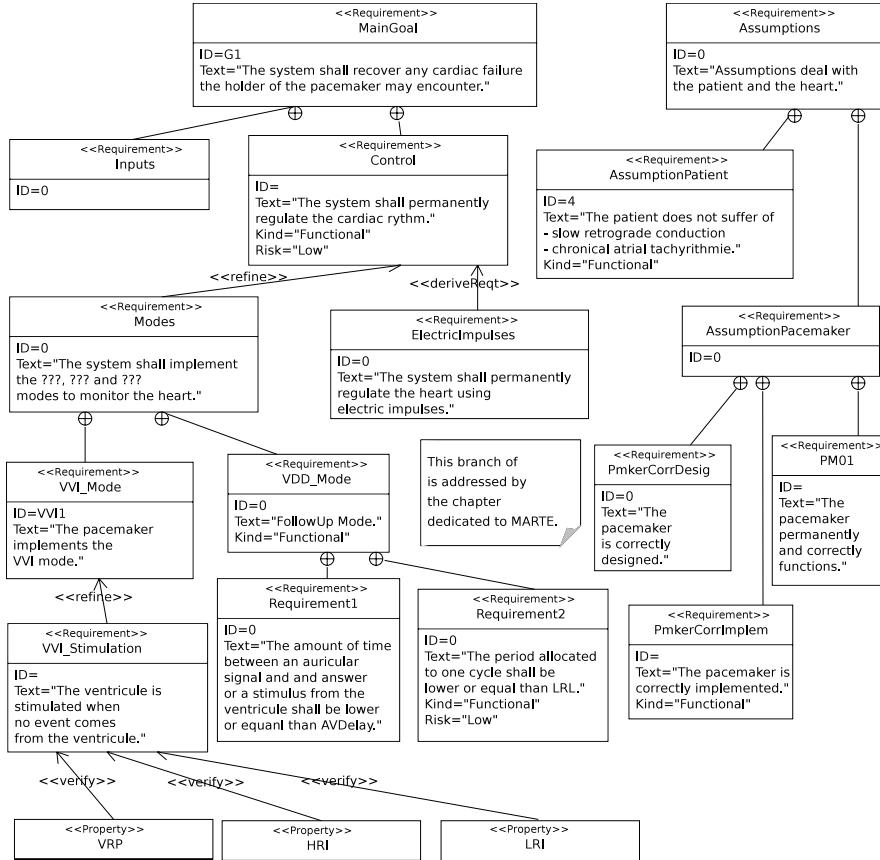


Figure 6.1 – Diagramme d'exigences : buts, hypothèses, exigences raffinées et propriétés

d'un modèle AVATAR nécessitent d'avoir un modèle « fermé ». C'est pourquoi le diagramme d'instances de blocs de la figure 6.2 décrit non seulement l'architecture interne du pacemaker mais aussi les connexions de celui-ci avec le cœur qu'il observe et stimule. Notons bien que la figure 6.2 décrit une architecture organique qui, dans une démarche d'ingénierie système, vient après l'architecture fonctionnelle non décrite dans ce chapitre car exposée précédemment dans le chapitre 5.

Le langage AVATAR réunit en un seul diagramme les diagrammes de définition des blocs et de vue interne des blocs pour former un diagramme d'instances de blocs. Les connecteurs qui relient les ports de deux instances de blocs permettent de définir des communications synchrones entre les machines à états locaux à ces deux instances.

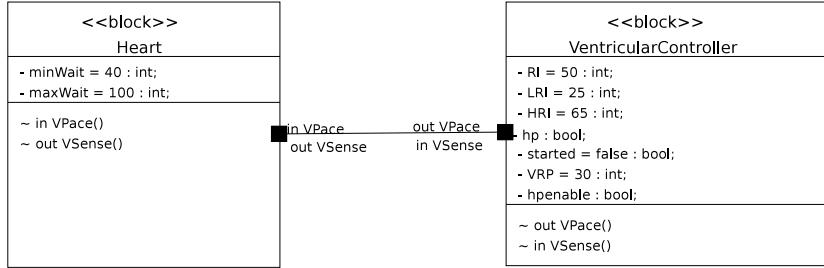


Figure 6.2 – Extrait du diagramme d'instances de blocs

6.5. Comportement

Toutes les instances de blocs représentées sur la figure 6.2 ont un comportement décrit par une machine à états. Nous avons choisi de présenter celles du contrôleur de ventricule (*VentricularController*) et du cœur dans les figures 6.3 et 6.4. Le premier diagramme (figure 6.3) met en évidence deux états principaux (*WaitRI* et *WaitVRP*) correspondant respectivement à l'exécution d'un impact sur le cœur si ce dernier n'a pas effectué son action normale, et à l'attente du prochain cycle.

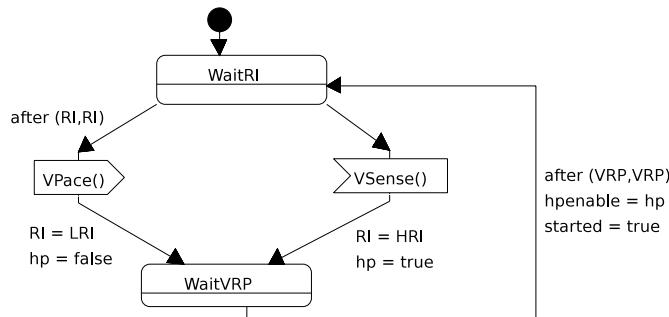


Figure 6.3 – Machine à états du contrôleur de ventricule

Les instances de blocs étant dotées de comportements, il devient possible de les simuler avec l'outil TTool. Celui-ci permet de jouer, pas à pas ou en rafale, des scénarios d'exécution du modèle. Les portions de modèle explorées sont rapidement identifiables sur les diagrammes eux-mêmes et les traces de simulation prennent la forme de diagrammes de séquences, tels que celui présenté à la figure 6.5. Ces traces mettent en évidence les communications synchrones et asynchrones entre les instances, les actions logiques effectuées par les instances (c'est-à-dire les changements de valeur des variables) et enfin les délais entre actions ou communications. Le simulateur de l'outil TTool gère une horloge unique, globale et commune à toutes les instances de

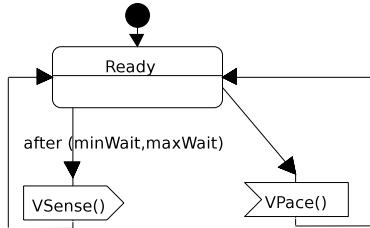


Figure 6.4 – Machine à états du contrôleur du cœur

blocs qui composent l'architecture du système. L'unité de temps est globale au système et correspond, par exemple, à une seconde. La notation « @50 » qui apparaît sur le tracé de simulation de la figure 6.5 exprime la date absolue dont l'occurrence a lieu cinquante unités de temps après le début de la simulation. Les nombres associés à des rectangles allongés dans le sens vertical spécifient quant à eux des progressions du temps.

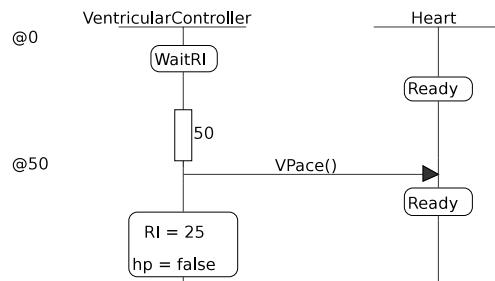


Figure 6.5 – Extrait du diagramme de séquences obtenu à partir d'une simulation du modèle du système

6.6. Vérification formelle du mode VVI

La pratique de la vérification formelle conduit à distinguer des propriétés qualifiées de « générales » applicables à une large classe de systèmes (section 6.6.1) et les propriétés spécifiques au système en cours d'étude (section 6.6.2).

6.6.1. Propriétés générales

Comme la plupart des systèmes temps réel, le pacemaker doit satisfaire des propriétés générales telles que l'absence de blocage (*deadlock*), l'absence de cycle interne sans échappatoire (*livelock*) et la capacité à retourner à l'état initial.

La vérification de ces propriétés à l'aide de l'outil UPPAAL nécessite la traduction du modèle AVATAR en automates temporisés. Cette dernière est entièrement prise en charge par TTool qui propose, en natif, de vérifier l'absence de blocage. La copie d'écran de la figure 6.6 montre ainsi que le modèle AVATAR du pacemaker ne comporte pas de *deadlock*. Cette bonne propriété a été vérifiée sans avoir à écrire ou lire une ligne du « code automate temporisé » automatiquement généré par TTool et injecté dans l'outil UPPAAL.

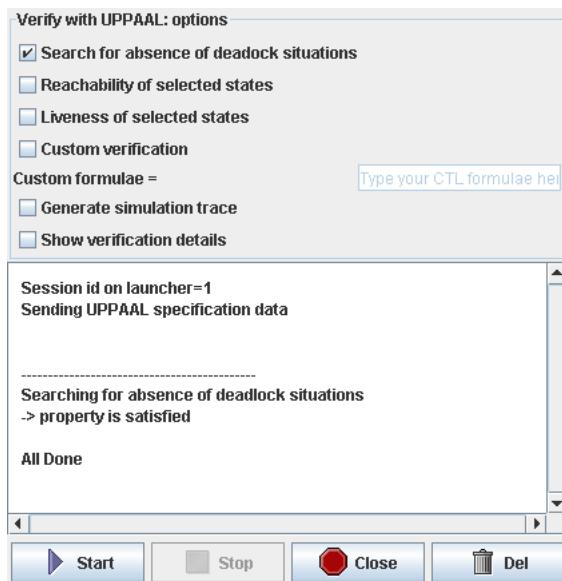


Figure 6.6 – Fenêtre de vérification : recherche de *deadlock*

6.6.2. Expression de propriétés en TEPE

Nous avons modélisé en TEPE les trois propriétés définies lors de la phase de modélisation des exigences (voir section 6.3.2) :

- LRI : lorsque le *pace* sur l'hystérésis est désactivé, un *pace* sur le ventricule, ou alors une action *sense*, du cœur doit être effectué avant LRI unités de temps ;
- HRI : lorsque le *pace* sur l'hystérésis est activé, un *pace* sur le ventricule, ou alors une action *sense*, du cœur doit être effectué avant HRI unités de temps ;
- VRP : une action *sense* ne peut être effectuée avant que VRP unités de temps ne se soient écoulées après l'action précédente *sense* ou une action *pace*.

A titre d'exemple, la figure 6.7 représente la propriété *LRI*. L'opérateur *TC* représente une contrainte de temps (*Time Constraint*) d'une valeur *LRI*. Ainsi, une fois le l'hystérésis désactivé, une « Action » (c'est-à-dire soit un signal *V Pace* soit un signal *V Sense*, voir l'opérateur « *Alias* ») doit être effectuée dans le système avant *LRI* unités de temps. D'une façon assez similaire, il est possible de modéliser graphiquement et simplement la propriété *HRI*.

La propriété *PropVRP* est plus complexe à modéliser car elle fait référence à des états particuliers de la machine à états du contrôleur. Ainsi, sa modélisation repose sur l'utilisation des signaux d'entrée dans les états *WaitVRP* et *WaitRI* (voir figure 6.8). La propriété modélise deux sous-propriétés mises en relation avec l'opérateur *OR*. Une sous-propriété exprime le cas où le système n'a pas encore effectué un premier cycle c'est-à-dire, *started* vaut *false*. La deuxième sous-propriété s'applique uniquement dans le cas où un premier cycle a été effectué : ainsi, le système a déjà atteint l'état *WaitVRP*, ce qui se traduit par le signal *enterState_WaitVRP*. Dans ce cas, le système doit attendre une durée *VRP* avant d'effectuer le cycle suivant dont le démarrage se traduit par l'entrée dans l'état *WaitRI* (signal TEPE *enterState_WaitRI*).

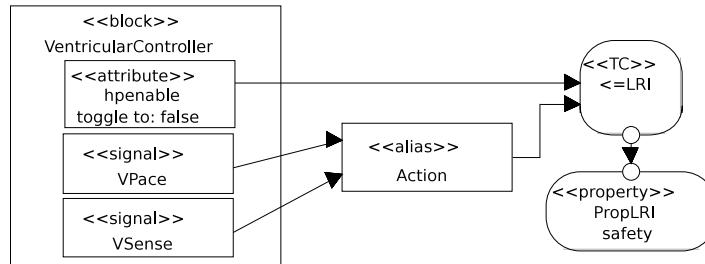


Figure 6.7 – Modélisation en TEPE d'une propriété de sûreté : *LRI*

La vérification des propriétés précédemment décrites repose soit sur leur expression sous la forme d'une formule en logique temporelle (section 6.6.3), soit par utilisation d'observateurs (section 6.6.4).

6.6.3. Utilisation de logiques temporelles

Une approche très usitée de vérification consiste à exprimer le système dans un langage formel (par exemple, LOTOS), à exprimer les propriétés à vérifier dans une logique temporelle (par exemple, CTL) puis à donner en entrée d'un *model-checker* à la fois la spécification du système et les propriétés. Le système est transformé en un système de transitions étiquetées sur lequel les propriétés sont étudiées par le *model-checker*. Parmi les logiques temporelles usuelles, nous pouvons citer LTL, CTL, PSL/-Sugar.

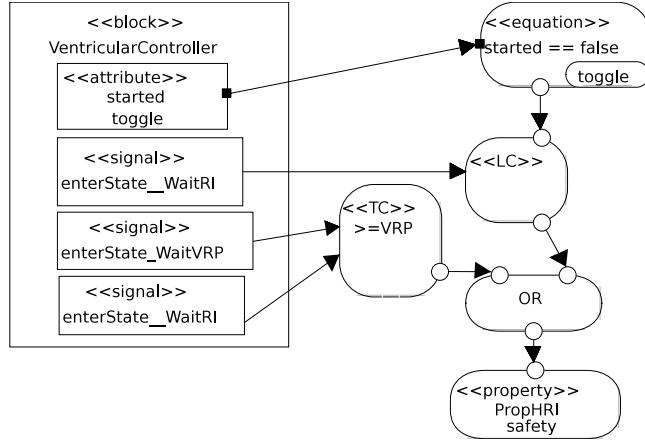


Figure 6.8 – Modélisation en TEPE d'une propriété de sûreté : PropVRP

Avec TTool, il est possible de saisir une formule CTL qui est analysée par le *model-checker* d'UPPAAL. Cette formule fait directement référence à l'automate d'UPPAAL, et nécessite donc d'ouvrir les automates générés par TTool avec UPPAAL, et de comprendre ces derniers. Afin de montrer la différence entre une propriété saisie graphiquement avec TEPE et une formule CTL correspondante, la formule CTL, au format UPPAAL, de la propriété LRI est :

```
A[] VentricularController__0.hp imply
VentricularController__0.h <= VentricularController__0.HRI
```

et celle de la propriété PropVRP est :

```
A[] ((VentricularController__0.id0 && VentricularController__0.started) imply
VentricularController__0.h_>= VentricularController__0.VRP
```

Aussi, la figure 6.9 montre la fenêtre de saisie d'une formule CTL et sa vérification depuis TTool.

6.6.4. Vérification guidée par des observateurs

La vérification guidée par les observateurs consiste à modifier le modèle de conception avec de nouveaux blocs qui sont chargés d'observer le modèle et d'effectuer une action particulière (par exemple, aller dans un état nommé « erreur ») si la ou les propriétés observées ne sont pas satisfaites. Les blocs d'observation reçoivent ainsi des signaux émis par les blocs de la conception, et prennent les décisions de satisfiabilité en fonction de ces signaux. Quelques remarques :

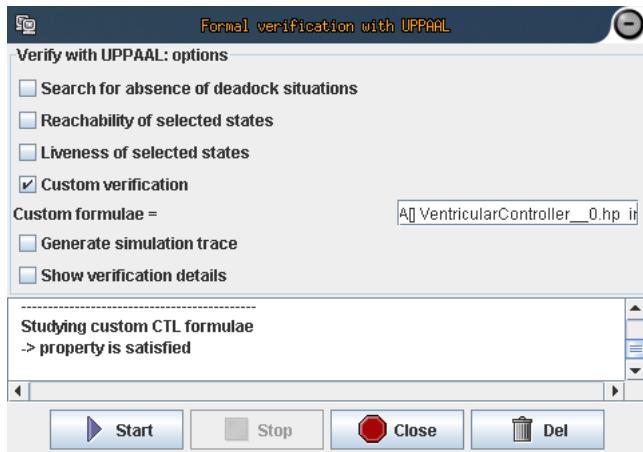


Figure 6.9 – Vérification d'une formule CTL depuis TTool : vérification de LRI

– les blocs d’observation sont par hypothèse non intrusifs sur le système observé, c’est-à-dire que leur adjonction au modèle ne doit en aucun cas modifier le résultat des propriétés observées ;

– l’adjonction de blocs d’observation peut parfois amener à modifier la conception afin d’ajouter des signaux émis par les blocs observés. Ces adjonctions de signaux se fait toujours en respectant l’aspect non intrusif de l’observateur, c’est-à-dire que l’observateur doit être prêt à accepter en permanence les signaux synchrones ajoutés au système pour l’observation.

A titre d’exemple, la figure 6.10 présente le diagramme d’instances de blocs auquel ont été ajoutés des observateurs pour les trois propriétés étudiées dans ce chapitre. Pour chaque nouveau bloc, il faut bien entendu fournir une machine à états.

La machine à états de l’observateur qui étudie les propriétés LRI et HRI est représentée à la figure 6.11. Les signaux émis par le contrôleur du ventricule sont analysés afin de déterminer si un intervalle de temps clairement spécifié s’écoule entre des réceptions, ou non. Cet intervalle de temps est modélisé par le démarrage, l’expiration et l’arrêt d’un *timer* qui sert de chronomètre.

TTool permet de rechercher automatiquement l’atteignabilité (*reachability*) des états ou actions des machines à l’état. La méthodologie de vérification à l’aide d’observateurs repose sur la preuve de non-atteignabilité des états d’erreur des observateurs. Cette preuve est menée automatiquement par TTool, en s’appuyant encore une fois sur la transformation automatique du modèle et des observateurs en automates temporisés, puis l’appel au *model-checker* d’UPPAAL avec des formules CTL obtenues

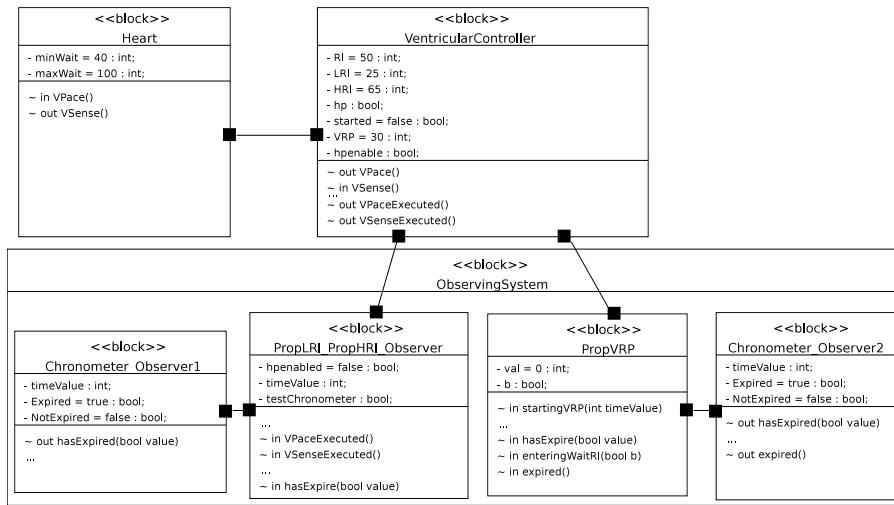


Figure 6.10 – Conception avec observateurs

par analyse de la modélisation SysML et de sa transformation en automates temporisés. La figure 6.11 montre le résultat de cette méthodologie de vérification appliquée au pacemaker, en particulier le fait que l'état d'erreur de l'observateur des propriétés LRI et HRI n'est pas atteignable. Avec un ordinateur récent, ce résultat de vérification est obtenu en quelques secondes : cela s'explique bien sûr par le fait que nous avons considéré seulement un sous-ensemble du modèle.

Les observateurs ont été ajoutés manuellement à la conception. En effet, actuellement, les propriétés TEPE ne peuvent pas être automatiquement transformées en observateurs. Nous travaillons actuellement à l'ajout à TTool de cette génération automatique, comme cela est le cas pour d'autres profils : TURTLE [FON 08] et DIPLODOCUS [KNO 11b].

6.6.5. Remontée au modèle

Lorsque l'on effectue des simulations ou des vérifications d'un modèle, il se pose la question de l'interprétation des résultats issus des dites simulations et vérifications par rapport au modèle fourni en entrée au simulateur ou au vérificateur.

Les simulations avec TTool de modélisations AVATAR sont effectuées directement sur les diagrammes. En particulier, TTool anime directement les machines à états du système simulé, et produit en outre un diagramme de séquences dont les noms des entités et des actions sont exactement ceux du modèle.

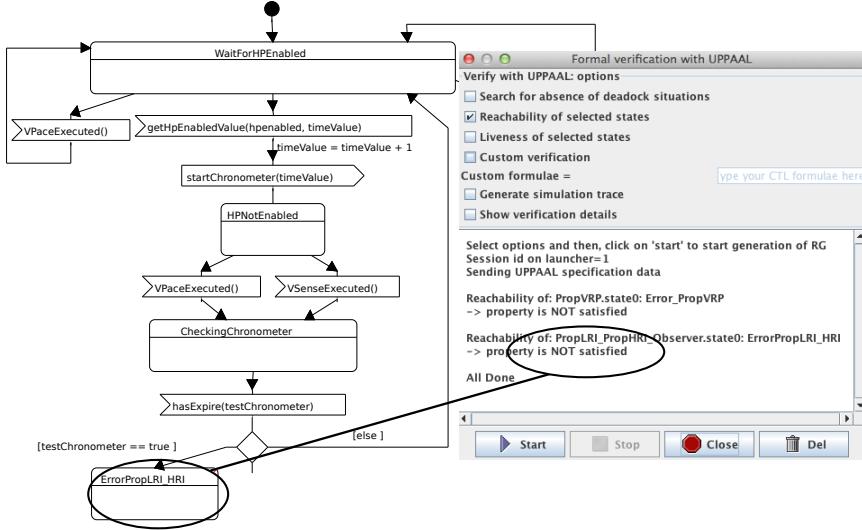


Figure 6.11 – Machine à états d'un observateur, et vérification de la non-accessibilité de l'état d'erreur de cet observateur

Dans le cas des vérifications, les preuves sont effectuées sur un modèle sous forme d'automates temporisés qui est structurellement différent du modèle AVATAR correspondant. Dans le cas d'une propriété non vérifiée, le *model-checker* d'UPPAAL exhibe une trace menant à la violation de la propriété. Actuellement, cette trace de simulation doit être analysée sous UPPAAL directement, TTool n'étant pas capable de la transformer en une trace AVATAR. La résolution de ce problème fait partie de nos travaux futurs.

6.7. Positionnement par rapport à d'autres travaux

Notre contribution consiste en un environnement de modélisation (AVATAR) et en un outil support à ce langage (TTool). Nous discutons du positionnement de ces deux contributions dans cette section.

6.7.1. Langages

Dans la mouvance des travaux qui jettent des ponts entre le langage SysML et les méthodes formelles supportées par des outils de vérification de modèles, le langage AVATAR précise la sémantique des diagrammes de conception SysML et intègre l'expression de propriétés directement avec un diagramme SysML là où d'autres environnements délèguent cette fonction de modélisation à un formalisme externe, logique

temporelle par exemple. Le langage d'expression de propriétés TEPE se veut plus accessible aux personnes rebutées par les langages formels que ne peut l'être le langage CCSL associé au profil UML MARTE.

Au-delà des langages de la famille UML, bien d'autres environnements permettent la modélisation d'un système temps réel et sa vérification formelle. UPPAAL [BEN 04] en est un bon exemple : il permet de modéliser un système sous la forme d'un ensemble d'automates communicants temporisés. La vérification formelle de propriétés CTL est directement intégrée à l'environnement du même nom (UPPAAL). En revanche, le langage sous forme d'automates peine à supporter un processus d'ingénierie système, et n'est pas usité non plus pour documenter un projet. Ce constat s'étend à d'autres formalismes tels que les réseaux de Petri supportés par l'outil TINA [BER 06], ou encore l'algèbre de processus LOTOS supportée par l'outil CADP [GAR 07].

6.7.2. Outils

UML/SysML vers ESL [VIE 06] est un environnement de modélisation pour les systèmes sur puces, offrant de la simulation et de la vérification formelle depuis les modèles constitués de diagrammes de séquences UML. Les résultats de simulation et de vérification permettent notamment d'obtenir les pires cas d'exécution. Mais là encore, la phase d'ingénierie n'est pas intégralement couverte. De plus, l'environnement est très limité en termes de communications.

L'environnement basé SysML présenté dans [SIL 09] permet la vérification – avec UPPAAL – de propriétés logiques et temporelles sur des modèles réalisés avec l'outil Rhapsody. Cette approche ne supporte pas la phase de recueil d'exigences. En l'absence de langage de haut niveau pour exprimer les propriétés, il faut utiliser des formules CTL. TTool permet au contraire de vérifier l'accessibilité et la vivacité d'actions en ajoutant par de simples clics de souris des marqueurs aux actions concernées.

L'environnement OMEGA [OBE 09] supporte la phase de capture d'exigences en raison de son support de SysML *via* Rhapsody, mais ne permet pas d'intégrer ces exigences (et des propriétés) dans l'approche de vérification formelle. OMEGA et TTool ne supportent pas les flux continus à la SysML, contrairement à l'outil Artisan, qui supporte de plus les probabilités sur les transitions, ainsi que les régions interruptibles.

Développée dans l'environnement Eclipse récemment enrichi avec l'éditeur Papyrus, l'atelier TopCased [FAR 06] fédère des outils d'analyse de modèle et des générateurs de code construits qui exploitent l'approche « transformation de modèle ». L'outil TTool utilisé dans ce chapitre repose sur une approche différente reposant sur un environnement moins flexible mais formalisé, tant du point de vue conception qu'expression de propriétés.

La recherche de nouvelles techniques de conception en électronique a aussi motivé des travaux liant SysML et les méthodes formelles. Ainsi, des modèles SysML peuvent être traduits en VHDL-AMS [Rea 11] ou en *simulink* [VAN 06]. En génie mécanique, SysML est aussi utilisé conjointement avec d'autres langages spécifiques tels que Modelica [PAR 10]. Ce langage décrit des composantes mécaniques, électriques, hydrauliques et thermiques d'un système sous la forme d'un ensemble d'équations qu'un simulateur peut résoudre à chaque pas temporel.

6.8. Conclusion

Un modèle SysML ne saurait rester au stade du dessin industriel. Un outil d'analyse de modèles SysML qui allie simulation et vérification formelle apporte des réponses au besoin de détecter des erreurs de conception au plus tôt dans le cycle de vie du système. Ce type d'outil confronte un modèle SysML à des exigences et des propriétés qui se prêtent à travailler avec des versions étendues du langage SysML. Tel est le cas de l'outil TTool utilisé dans ce chapitre pour confronter un modèle du pacemaker exprimé dans le langage SysML/AVATAR à des propriétés de sûreté.

A partir du modèle SysML proposé au chapitre 5 de cet ouvrage, nous avons construit un modèle AVATAR orienté vers l'analyse des comportements logiques et temporels du contrôleur du pacemaker. L'étude du mode VVI nous a permis d'illustrer l'usage d'AVATAR et TTool sur l'étendue de ce chapitre. La simulation à titre de débogage, la vérification par contrôle de modèle et la vérification guidée par des observateurs, ont été successivement présentées.

Ces trois modes d'analyse de modèles complémentaires nous ont permis de prouver formellement la satisfaction de plusieurs exigences. Ces résultats de simulation et de vérification reposent sur le modèle fourni par l'utilisateur dans TTool, sur l'absence d'erreurs dans le processus de traduction vers des automates temporisés, et sur l'absence d'erreurs dans le *model-checker* d'UPPAAL. L'utilisation de traducteur d'outils de vérification certifiés ou prouvés permettrait d'asseoir sur des bases plus solides les résultats de vérification.

6.9. Pour aller plus loin

Pour utiliser TTool, il suffit de se rendre sur le site Internet ttool.telecom-paristech.fr puis de télécharger l'outil qui fonctionne sous Windows, MacOS et Linux. Une fois l'outil installé, il suffit de double-cliquer sur *ttool.jar*.

TTool intègre un éditeur graphique, un simulateur de modèles et des générateurs de code dédiés aux outils externes. Dès le premier clic on accède à l'éditeur et au simulateur. Si les outils externes ont été téléchargés et le fichier de configuration paramétré

en conséquences alors la vérification formelle peut être pilotée depuis les menus de TTool en faisant référence aux identificateurs du modèle AVATAR (et non au code généré à partir de celui-ci). Le site *web* dédié à l'outil et au profil AVATAR propose un tutoriel d'utilisation d'AVATAR, des exemples de modèles et présente à la fois le langage AVATAR et les techniques de vérification disponibles.

L'éditeur de diagrammes supporte AVATAR, déclinaison de SysML dotée d'extensions temps réel et d'une sémantique formelle. Le simulateur intégré à TTool facilite la mise au point et le débogage de modèles. Les interfaces aux outils de vérification formelle UPPAAL et ProVerif autorisent des analyses de modèles plus poussées sans requérir pour autant une maîtrise des méthodes formelles. Ces deux outils externes ciblent respectivement les propriétés de sûreté et de sécurité.

6.10. Bibliographie

- [ABA 02] ABADI M., BLANCHET B., « Analyzing Security Protocols with Secrecy Types and Logic Programs », *29th Annual ACM SIGPLAN - SIGACT Symposium on Principles of Programming Languages (POPL 2002)*, Portland, Oregon, ACM Press, p. 33–44, janvier 2002.
- [APV 11a] APVRILLE L., SAQUI-SANNES P. D., « AVATAR/TTool : un environnement en mode libre pour SysML temps réel », *Génie Logiciel*, vol. 98, p. 22–26, septembre 2011.
- [APV 11b] APVRILLE L., « webpage of TTool », ttool.telecom-paristech.fr/, 2011.
- [BEN 04] BENGSSON J., YI. W., « Timed Automata : Semantics, Algorithms and Tools », *Lecture Notes on Concurrency and Petri Nets*, W. Reisig and G. Rozenberg (eds.), LNCS 3098, Springer-Verlag, p. 87-124, 2004.
- [BER 06] BERTHOMIEU B., VERNADAT F., « Time Petri Nets Analysis with TINA », *3rd Int. IEEE Conf. on The Quantitative Evaluation of Systems (QEST 2006)*, p. 123-124, 2006.
- [FAR 06] FARAIL P., GAUFILLET P., CANAL A., CAMUS C. L., SCIAMMA D., MICHEL P., CRÉGUT X., PANTEL M., « The topcased project : a toolkit in open source for critical aeronautic systems design », *In ERTS2006 : Embedded Real Time Software*, Toulouse, France, novembre 2006.
- [FON 08] FONTAN B., Méthodologie de conception de systèmes temps réel et distribués en contexte UML/SysML, Thèse de doctorat, Laboratoire d'Analyse et d'Architecture des Systèmes (LAAS-CNRS), 2008.
- [GAR 07] GARAVEL H., LANG F., MATEESCU R., SERWE W., « CADP 2006 : A Toolbox for the Construction and Analysis of Distributed Processes », *Computer Aided Verification (CAV'2007)*, vol. 4590, Berlin Germany, p. 158-163, 2007.
- [Kno 11a] KNORRECK D., APVRILLE L., DE SAQUI-SANNES P., « TEPE : A SysML Language for Time-Constrained Property Modeling and Formal Verification », *ACM SIGSOFT Software Engineering Notes*, vol. 36/1, p. 1–8, janvier 2011.

- [KNO 11b] KNORRECK D., UML-based Design Space Exploration, Fast Simulation and Static Analysis, PhD thesis, EDITE, Ecole Doctorale Informatique, Télécommunications et Electronique, Telecom ParisTech, 2011.
- [KOY 92] KOYMANS R., *Specifying Message Passing and Time-Critical Systems with Temporal Logic*, Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1992.
- [LEB 10] LEBLANC P., « Modélisation SysML exécutable d'un système événementiel et continu », *Livre blanc IBM, files.me.com/jmbruel/gblurt*, 2010.
- [LET 05] LETIER E., KRAMER J., MAGEE J., UCHITEL S., « Fluent temporal logic for discrete-time event-based models », *Proceedings of the 10th European software engineering conference, ESEC/FSE-13*, New York, NY, USA, ACM, p. 70–79, 2005.
- [OBE 09] OBER I., DRAGOMIR I., « OMEGA2 : A new version of the profile and the tools », *14th IEEE International Conference on Engineering of Complex Computer Systems, UML-AADL'2009*, Potsdam, p. 373-378, 2009.
- [OMG 11] OMG O. M. G., « SysML », www.sysml.org/, 2011.
- [PAR 10] PAREDIS C. J., BERNARD Y., BURKHART R. M., DE KONING H.-P., FRIEDENTHAL S., FRITZSON P., ROUQUETTE N. F., SCHAMAI W., « An Overview of the SysML-Modelica Transformation Specification », *INCOSE'2010*, 2010.
- [Rea 11] REALTIME-AT-WORK, « SysML-Companion : Virtual prototyping from SysML models », www.realtimeatwork.com/software/sysml-companion/, 2011.
- [SAT 10] SATURN P. F., « SATURN, SysML bAsed modeling, architecTure exploRation, si- mulation and syNthesis for complex embedded systems », www.saturn-fp7.eu/, 2010.
- [SIL 09] DA SILVA E. C., VILLANI E., « Integrating SysML and model-checking techniques for the v&v of space-based embedded critical software (In Portuguese) », *Brasilian Symposium on Aerospace Engineering and Applications*, 2009.
- [VAN 06] VANDERPERREN Y., DEHAENE W., « From UML/SysML to Matlab/Simulink : current state and future perspectives », *DATE '06 : Proceedings of the conference on Design, automation and test in Europe*, 3001 Leuven, Belgium, Belgium, European Design and Automation Association, p. 93–93, 2006.
- [VIE 06] VIELHL A., SCHONWALD T., BRINGMANN O., ROSENSTIEL W., « Formal performance analysis and simulation of UML/SysML models for ESL design », *DATE '06 : Proceedings of the conference on Design, automation and test in Europe*, Munich Allemagne, p. 1-6, 2006.

TROISIÈME PARTIE

MARTE

Chapitre 7

Présentation des concepts de MARTE

7.1. Introduction

Dans le cadre de ses activités dans le domaine de l'ingénierie des modèles, l'OMG a normalisé une extension de UML pour le domaine du temps réel et de l'embarqué, MARTE. L'objet de ce chapitre est de tracer les grandes lignes de cette norme de façon à fournir l'information nécessaire pour explorer plus en avant cette norme. De plus, on reviendra plus en détail sur certains points particuliers, comme le support de MARTE à une modélisation à base de composant d'une application et, dans ce contexte, nous expliquerons comment MARTE permet de modéliser à ce niveau des propriétés telles que la concurrence ou le temps. Nous verrons également un autre aspect important du domaine du temps réel embarqué qui est la prise en compte de la modélisation des plates-formes, avant de passer en revue les facilités de MARTE quant à l'analyse quantitative des modèles, et plus particulièrement l'analyse d'ordonnançabilité.

Cependant, l'objet de cette section n'est pas d'être un manuel utilisateur de MARTE. L'objectif est plutôt de fournir aux lecteurs le minimum d'informations nécessaires pour entrer de façon efficace dans l'analyse et l'utilisation de cette norme.

7.2. Généralités

En septembre 2003, l'OMG publia sa première norme de modélisation dans le domaine du temps réel : le profil nommé SPT (*UML Profile for Schedulability, Performance, and Time*) [OMG 05a]. Le domaine d'application de SPT était restreint à

Chapitre rédigé par Sébastien GÉRARD et François TERRIER.

l’analyse quantitative de modèles UML. Il permettait en particulier d’annoter des modèles UML pour effectuer des analyses d’ordonnançabilité s’inscrivant dans le cadre des techniques de type RMA (*Rate Monotonic Analysis*) et des analyses de performance conformes à des techniques basées sur la théorie des files d’attente.

Peu après sa sortie, des industriels et des chercheurs ont publié des travaux sur cette norme et plus particulièrement sur son adéquation vis-à-vis des pratiques industrielles et des pratiques académiques, dans le cadre de l’ingénierie des modèles, en particulier dans le domaine du développement par les modèles des systèmes temps réel embarqués [GéR 03, CEA 05]. Entre autres, on peut citer le CEA, l’INRIA et Thales qui au travers de plusieurs projets réalisés dans le cadre du programme de recherche commun, CARROLL, ont été les principaux acteurs de la définition et de la publication d’une nouvelle norme, MARTE, destinée à remplacer SPT et couvrir un domaine d’application plus large : le domaine du développement par les modèles des systèmes temps réel embarqués. Il s’agissait notamment de couvrir des besoins particuliers en termes de modélisation des plates-formes logicielles et matérielles et de supporter des modélisations par composant [OMG 05b].

La première version de la norme MARTE a été publiée par l’OMG en novembre 2009 (la version en cours à cette date est la version 1.2). Notons que toutes les normes publiées par l’OMG sont publiques et disponibles gratuitement sur le site de l’OMG¹.

Notons que toutes les normes publiées par l’OMG suivent un processus de maintenance défini dans [OMG 09a]. Entre autres, chaque norme est mise à jour pour corriger les problèmes levés par ses utilisateurs et implanteurs aux travers de groupes dédiés de révision nommés RTF (*Revision Task Force*). MARTE est dans son troisième cycle de révision qui devrait aboutir en 2013 et donner lieu à la version 1.3 de la norme.

7.2.1. Les usages possibles de MARTE

La clause 6.2.3 de la norme MARTE définit un certain nombre de cas d’utilisation attendus, incluant les acteurs sous-jacents. On y trouve deux catégories principales d’utilisateurs de la norme, à savoir les méthodologues et fournisseurs d’infrastructures d’exécution d’un côté, les utilisateurs du langage de l’autre.

La première catégorie correspond à ceux qui, s’appuyant sur MARTE, vont en définir un usage correspondant aux besoins d’un domaine spécifique, fournir un support à l’exécution des modèles résultants, ou encore en spécialiser ou en étendre une partie vis-à-vis d’une technologie ou d’un domaine particulier.

1. Voir www.omg.org/spec/.

La seconde catégorie désignent ceux qui vont mettre en œuvre les résultats issus du travail des participants de la première catégorie.

REMARQUE. Il est important de noter que, à l'instar d'UML, MARTE est un langage de modélisation, et qu'en tant que tel, sa spécification n'en détaille pas un usage particulier. Il est du ressort des membres de la première catégorie d'usage précédemment introduite de fournir cette information aux ingénieurs, utilisateurs de la norme.

Les chapitres qui suivent dans cette partie illustrent certains usages de la norme sur l'exemple du pacemaker : le chapitre 8 montre comment utiliser MARTE pour modéliser l'application du pacemaker, le chapitre 9 illustre les possibilités de validation sur ce même modèle. Enfin, le chapitre 10 illustre la conception orientée composants à partir d'un modèle de type MARTE et la génération de code.

7.2.2. Comment lire la norme ?

Le domaine couvert par la norme MARTE est vaste, puisqu'elle doit répondre aux besoins liés à la modélisation pour la conception et la validation des systèmes temps réel embarqués, et ceci en prenant en compte le plus grand nombre possible de technologies. Par conséquent, le document de spécification de MARTE est de taille importante, puisqu'il contient presque 800 pages. Afin d'en rendre pratique l'usage, sa conception a été pensée de façon modulaire.

La norme contient la définition de cas de conformité (voir clause 2 de [OMG 09b]) en permettant une utilisation à façon et optimale. La définition de ces cas de conformité s'appuie sur un ensemble de cas d'utilisation attendus de la norme. Les cas de conformité sont spécifiés à deux niveaux d'abstraction : basique et complet. Chaque niveau spécifie un ensemble concret d'unités d'extension considérées comme obligatoires pour le niveau considéré. Le niveau basique est un sous-ensemble du niveau complet et les unités d'extension du niveau complet sont considérés comme optionnelles au niveau basique.

MARTE a mis à profit la possibilité de décomposer un profil en une hiérarchie de sous-profil. L'intérêt de cette structuration est de séparer les préoccupations du profil et d'en faciliter ainsi la compréhension. Un autre avantage est que chaque sous-profil peut être utilisé indépendamment (avec éventuellement les sous-profil dont il dépend). Les unités d'extension des cas de conformité reflète ainsi l'architecture du profil MARTE et correspondent à ses différents sous-profil.

Par exemple, le cas de conformité intitulé « modélisation logicielle », dont le sujet est le support à la modélisation des aspects logiciels des applications temps réel embarquées incluant la prise en compte des propriétés non fonctionnelles, nécessite ainsi les sous-parties de MARTE suivantes :

- au niveau basique : « Generic resource Modeling », « Non-functional Properties », « Time » et « High-Level Application Modeling » ;
- au niveau complet : « Generic resource Modeling », « Non-functional Properties », « Value Specification Language », « Time », « Clock Handling Facilities », « Software Resource Modeling », « Generic Component Model » et « High-Level Application Modeling ».

Pour tous les détails sur le contenu de chaque cas de conformité, nous invitons les lectrices et les lecteurs à se reporter au tableau 2.2 de la clause 2.4.2 de la norme [OMG 09b].

Toujours dans le souci de faciliter son usage, l’organisation des chapitres décrivant chacun des sous-profil de MARTE se conforme au même patron décrit ci-après :

- 1) une première section délimitant les contours du sous-profil concerné et son architecture ;
- 2) une deuxième section décrivant le modèle de domaine relatif à la préoccupation du sous-profil, qui est accompagné d’un texte général présentant les grandes lignes du modèle. Cette section peut elle-même être découpée en sous-sections en fonction de la taille du domaine concerné. La description sémantique de chaque élément est consignée dans l’annexe F de la norme. Il n’est pas nécessaire de lire cette annexe en première lecture, pour revenir dessus ultérieurement lorsqu’il faut d’approfondir l’utilisation d’un sous-profil ;
- 3) une troisième section qui se décompose généralement elle-même en deux sous-sections :
 - a) une première sous-section qui contient l’ensemble des diagrammes de profils UML décrivant les extensions définies dans le sous-profil,
 - b) une seconde sous-section décrivant dans les détails chaque stéréotype du profil et incluant les règles de bonne construction des modèles relatives à l’utilisation du stéréotype, ainsi que d’éventuelles propositions d’extension de la notation UML ;
- 4) une quatrième section qui propose un ensemble d’exemples illustrant les extensions précédemment définies.

Pour conclure sur ce sujet, précisons que l’une des lignes de conduite ayant motivé les choix faits quant à la définition des concepts de MARTE fut de minimiser les efforts des utilisateurs pour sa mise en œuvre. Cependant, l’atteinte de cet objectif, notamment fournir des capacités adéquates et précises de modélisation pour le domaine du temps réel embarqué, nécessite un complément d’activité vis-à-vis d’une utilisation plus commune, car moins contrainte, d’UML.

7.2.3. L'architecture de MARTE

MARTE est un profil UML destiné à supporter une approche d'ingénierie dirigée par les modèles pour le développement de systèmes temps réel embarqués. Il se compose d'un ensemble d'extensions appropriées des concepts généraux d'UML offrant aux concepteurs des constructions de langage de première classe pour leur domaine d'application.

Beaucoup de ces extensions concernent ce que l'on appelle les aspects non fonctionnels des applications temps réel embarquées. Ces préoccupations peuvent être classées en deux catégories traitant respectivement des aspects quantitatifs et qualitatifs. En outre, elles peuvent être utilisées à différents niveaux d'abstraction et peuvent être appliquées aussi bien à la modélisation des applications qu'à leur analyse par les modèles, ou les deux à la fois. Afin de satisfaire toutes ces exigences, MARTE est structurée en une hiérarchie de sous-profil, comme le montre le diagramme de la figure 7.1.

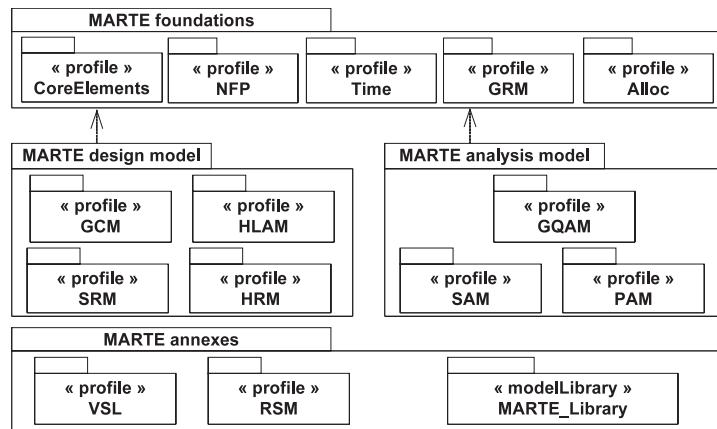


Figure 7.1 – Vue de l'architecture globale de MARTE.

Le paquetage « foundations » (situé en haut de la figure 7.1) définit les concepts de base de MARTE et se constitue des quatre sous-profil suivants :

- CoreElements : ce sous-profil propose essentiellement des concepts permettant de modéliser les modes opérationnels d'un système ;
- NFP (*Non-functional Properties*) et VSL (*Value Specification Language*) : le sous-profil NFP fournit les constructions de modélisation nécessaires pour déclarer, qualifier et appliquer à un modèle UML des informations non fonctionnelles. VSL est le compagnon indispensable de NFP puisqu'il permet de spécifier les valeurs des annotations non fonctionnelles définies dans le sous-profil NFP ;

– *Time* et CCSL (*Clock-Constraint Specification Language*) : le sous-profil *Time* définit le concept de temps, primordial pour les systèmes temps-réel embarqués. MARTE propose trois modèles de temps : un modèle de temps chronométrique, un modèle de temps logique et un modèle de temps correspondant aux approches de développement s'appuyant sur le paradigme de la programmation synchrone. Par ailleurs, ce sous-profil de MARTE définit un ensemble de mécanismes de bas niveau pour manipuler le temps, comme des horloges, des observateurs ou encore des événements temporisés. CCSL est un langage textuel complémentaire au profil *Time* permettant de décrire des contraintes entre des horloges. *Time* et CCSL ne sont pas détaillés ici ; les lecteurs qui voudront en savoir plus sur ce sujet sont invités à lire [ESP 09] ;

– GRM (*Generic Ressource Modeling*) : ce sous-profil satisfait une exigence importante pour le domaine du temps réel embarqué, à savoir la modélisation des plates-formes. Le concept de plate-forme a été abstrait dans la norme sous la forme de ressource et on parlera de ressource de communication ou de ressource de calcul par exemple. Le profil GRM contient ainsi une taxonomie de ressources permettant de modéliser tous les aspects d'une plate-forme à un niveau d'abstraction système, c'est-à-dire indépendamment des préoccupations logicielles et/ou matérielles. Ces deux aspects sont traités dans deux autres sous-profils spécialisés de MARTE, respectivement SRM (*Software Ressource Modelling*) et HRM (*Hardware Ressource Modelling*). En plus du support à la modélisation des plates-formes au niveau système, GRM permet également de modéliser l'usage de la plate-forme au travers de relations spécifiques liant éléments d'application et éléments de plate-forme ;

– Alloc (*Allocation Modeling*) : les cycle de développement couramment utilisés dans le domaine du temps réel embarqué reposent sur la modélisation explicite d'un modèle d'allocation permettant de mettre en relation les éléments du modèle de l'application avec les éléments du modèle de la plate-forme. Le sous-profil « Alloc » définit les concepts nécessaires à la description explicite de ce modèle d'allocation.

Les sous-profils qui viennent d'être décrits forment le socle conceptuel de MARTE. Sur la base de ce dernier sont définis d'autres concepts plus spécifiques, pour modéliser de façon plus détaillée des applications temps réel embarquées d'une part, et pour considérer des approches mettant en œuvre des analyses quantitatives des modèles d'autre part.

Le développement des systèmes temps réel embarqués à base de modèles est principalement supporté de façon déclarative avec MARTE, c'est-à-dire sur la base d'annotations associées aux éléments d'un modèle précisant ainsi leurs caractéristiques. Les quatre sous-profils de MARTE dédiés à cette préoccupation sont :

– HLAM (*High-Level Application Modeling*) : ce sous-profil fournit un ensemble d'extensions à UML permettant d'annoter les éléments d'un modèle de façon à définir les propriétés TRE d'un système. On peut ainsi, par exemple, modéliser la concurrence ou préciser un modèle de calcul particulier ;

– GCM (*Generic Component Model*) : le paradigme de composant est essentiel pour modéliser les systèmes temps réel embarqués, et en particulier leur architecture. Pour ce faire, GCM reprend les structures composites d'UML et en étend un sous-ensemble afin de répondre au mieux aux besoins du domaine en matière de modélisation à base de composants. Les particularités principales de MARTE sont d'en préciser la sémantique, notamment le lien entre les éléments structurels et les éléments comportementaux. MARTE répond ainsi clairement à la question : que se passe-t-il quand un composant reçoit un stimulus sur l'un de ses ports ? D'autre part, le modèle de composant tel que défini dans MARTE supporte bien entendu le modèle client-serveur, et introduit la communication par échanges de données entre composants, à l'instar de SysML ;

– SRM (*Software Resource Modeling*) et HRM (*Hardware ressource Modeling*) : ces deux sous-profilspécialisent le sous-profil GRM pour fournir respectivement un support à la modélisation des plates-formes logicielles et matérielles. Sur la base de SRM, MARTE intègre nativement les modèles de trois types de plates-formes particulières, à savoir OSEK, ARINC et partiellement POSIX.

Précisons l'existence d'un autre sous-profil de MARTE plus particulièrement dédié à la modélisation des systèmes sur puce : le sous-profil RSM (*Repetitive Structure Modelling*). Ce dernier permet de décrire des systèmes sur puce de type calculs massivement parallèles, comme par exemple des systèmes implantant des algorithmes de traitement de l'image. RSM permet ainsi de décrire ce type de système et de prendre en compte leur structure généralement régulière et composée de beaucoup d'éléments, en fournissant des concepts favorisant la description compacte des modèles.

Enfin, sur la base des principes établis dans SPT, MARTE fournit un support étendu pour les approches d'analyse quantitative par les modèles des systèmes temps réel embarqués, en particulier les analyses d'ordonnançabilité et de performance. Les ajouts majeurs dans MARTE sont, d'une part, la définition d'un sous-profil d'analyse générique défini comme le sous-profil « foundation » pour l'analyse quantitative à base de modèles. D'autre part, le spectre des analyses permises par MARTE a été élargi, autant pour les analyses d'ordonnançabilité que pour les analyses de performance.

7.2.4. Au sujet de MARTE et SysML

SysML et MARTE sont des langages de modélisation construits sur la base de profils UML. De ce fait, ils peuvent être appliqués sur un même modèle rendant ainsi possible un usage commun de ces deux langages dédiés.

Il faut savoir que ces deux normes ont été spécifiées en parallèle sans effort de synchronisation. Cependant, de par leur périmètre respectif, l'ingénierie des modèles pour le domaine du temps réel embarqué et l'ingénierie système par les modèles, ces deux

normes sont complémentaires sur bien des aspects. Les capacités en termes de modélisation de ces deux normes couvrent un périmètre large permettant de satisfaire de nombreuses approches méthodologiques. Cela étant, si l'usage combiné des deux normes est utile dans de nombreuses situations, il n'en reste pas moins la difficulté de faire les bons choix de combinaison entre les concepts de MARTE et de SysML. Cette tâche est définitivement du ressort des méthodologistes évoqués ci-avant qui doivent définir un cadre architectural, consistant à utiliser au mieux les deux normes en fonction des contraintes du domaine visé et de ses préoccupations.

Ce n'est pas l'objet de ce chapitre, ni de ce livre, que de traiter de la question de définir un bon usage de MARTE et SysML. Pour aller plus loin sur ce sujet, les lectrices et lecteurs peuvent consulter [ESP 09] qui définit un certain nombre de cas d'utilisation conjointe des deux normes à partir desquels il est possible de redéfinir de nouvelles approches et de nouveaux usages.

7.2.5. *Un support open source*

Dans le cadre de ses travaux sur l'outil *open source* Papyrus² dédié à la modélisation UML2, le CEA LIST a développé une implantation *open source* de MARTE. Ce module complémentaire de Papyrus intègre l'ensemble des extensions définies dans la norme, l'ensemble des bibliothèques des modèles spécifiés dans ses annexes et propose un éditeur avancé textuel du langage de description des valeurs, VSL.

7.3. Quelques détails de MARTE

La suite de ce chapitre détaille quelques aspects de MARTE. Il ne s'agit pas de faire une revue systématique de tous les concepts, mais juste d'en présenter quelques-uns choisis de façon à donner aux lecteurs les bases pour une première utilisation de MARTE.

7.3.1. *Modéliser les propriétés non fonctionnelles*

On rappelle que l'objectif principal de MARTE est de définir un cadre général permettant d'annoter des modèles UML afin d'ajouter au modèle fonctionnel de l'application la description des propriétés dites non fonctionnelles, telles que des propriétés temps réel ou d'embarquabilité. Répondre à cette question nécessite en fait de s'intéresser aux questions corollaires suivantes :

- comment les propriétés non fonctionnelles doivent-elles être décrites, et en particulier quelles sont les différentes catégories à considérer ?

2. Voir www.eclipse.org/papyrus.

- comment associer des propriétés non fonctionnelles à un modèle UML ?
- quelles sont les relations possibles entre propriétés non fonctionnelles et comment les décrire ou/et les contraindre ?
- quel niveau d'abstraction faut-il pour faciliter l'utilisabilité de ce cadre général ?
- comment étendre le cadre normalisé, qui est forcément incomplet, de façon à couvrir de nouveaux domaines ou technologies ?
- comment décrire formellement les valeurs des propriétés non fonctionnelles ?

Notons que le terme « propriété non fonctionnelle » recouvre les termes de « qualité de service » ou encore de « propriétés extra fonctionnelles » que l'on peut rencontrer dans d'autres domaines (par exemple, dans les télécommunications).

Rappelons que le cadre proposé par MARTE pour résoudre les questions précédentes est constitué d'un profil dédié, NFP, d'un langage textuel, VSL, et de quelques bibliothèques de modèles dédiés et prédéfinis. NFP définit les concepts nécessaires à la description et à l'usage des propriétés non fonctionnelles. VSL permet de décrire avec précision les valeurs des propriétés non fonctionnelles, ainsi que certains qualificatifs comme l'unité d'une valeur ou sa précision. VSL ne sera pas abordé dans ce chapitre autrement qu'au travers d'exemples. Pour plus de détails sur sa sémantique et sa syntaxe, les lecteurs sont invités à se référer directement à l'annexe B de la norme MARTE. Enfin, les bibliothèques de modèles proposent un ensemble de propriétés non fonctionnelles prédéfinies et un certain nombre d'éléments de modèle considérés comme des utilitaires de MARTE.

NFP définit un ensemble minimal de concepts nécessaires pour implémenter la déclaration des propriétés non fonctionnelles et la spécification des relations pouvant exister entre ces propriétés non fonctionnelles. Cet ensemble contient cinq concepts que nous allons maintenant voir en détail.

Lorsque l'on veut donner du sens à des valeurs, il est important de pouvoir spécifier l'unité dans lesquels une valeur est exprimée. Pour ce faire, MARTE propose de définir des dimensions et leurs unités respectives. Spécifier une dimension consistera à modéliser un énuméré de UML annoté du stéréotype « Dimension » et ses littéraux seront eux annotés avec « unit ». La propriété principale de « Dimension » est « symbol ». C'est une chaîne de caractères qui dénote le symbole représentant la dimension, par exemple « T » pour le temps. Les propriétés de « unit » sont baseDimension et offsetValue. Ces deux propriétés permettent de spécifier respectivement pour chaque unité un facteur de conversion multiplicateur et une valeur de décalage par rapport à une unité particulière de la dimension définie comme l'unité de base. La figure 7.2 ci-après présente un exemple issu d'une bibliothèque de MARTE modélisant la dimension du temps et ses unités (par exemple, jour, heure, min, etc.).

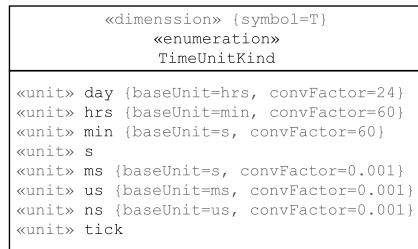


Figure 7.2 – Exemple de définition d'une dimension et de ses unités : TimeUnitKind

Les deux premiers concepts de sous-profil de modélisation des propriétés non fonctionnelles sont les concepts de « NFP » et « NfpType ».

« NFP » est utilisé pour marquer explicitement dans un modèle une propriété comme étant représentative d'une information non fonctionnelle.

La figure 7.3 illustre l'exemple de la définition d'une classe modélisant un processeur. Ce processeur comporte un identifiant (id) et un nom (name) et deux propriétés non fonctionnelles permettant de définir respectivement la vitesse du processeur (speed) et sa consommation électrique (powerConsumption).

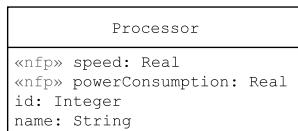


Figure 7.3 – Exemple d'utilisation du concept « nfp »

« NfpType » est utilisé pour déclarer des types de propriété non fonctionnelle. Ce stéréotype est une extension du concept de type de donnée de UML, « DataType ». Ce qui différencie un type de donnée UML d'une classe UML est qu'une instance d'un type de donnée est uniquement identifiée à travers sa valeur.

Avant de continuer plus avant dans la description des concepts du sous-profil NFP, faisons une parenthèse sur le concept de stéréotype et plus précisément sur les styles de conception des stéréotypes.

Dans les profils UML, on peut rencontrer deux formes de stéréotype (les deux formes n'étant pas exclusives) : d'un côté, l'usage commun des stéréotypes consiste à les utiliser pour modéliser de façon déclarative des métadonnées augmentant ainsi la sémantique d'un modèle UML. Par exemple, on peut annoter une opération d'une classe

de façon à spécifier son temps d'exécution ou sa taille en mémoire. Dans ce cas, le typage des propriétés d'un stéréotype consiste en un type primitif, un type énuméré, un type de donnée structurée ou une classe.

La limitation principale de cette première façon de faire provient du fait que les types des propriétés des stéréotypes doivent être connus à l'avance. Ainsi, les types utilisés doivent être préalablement définis dans le profil ou dans une bibliothèque de modèle possédée ou importée par le profil. Si pour une raison ou une autre, un utilisateur souhaite modifier une propriété existante pour adapter son type à un domaine/besoin spécifique, la seule solution est de redéfinir le profil et d'introduire de nouvelles caractéristiques ou de modifier les caractéristiques existantes des stéréotypes.

Pour permettre une définition plus flexible des stéréotypes et donc d'un profil, l'idée est de définir les caractéristiques des stéréotypes par référence. Pour ce faire, il s'agit de typer les caractéristiques d'un stéréotype par une métaclasses de UML2. Ce faisant, l'application du stéréotype sur un élément de modèle utilisateur, permet de donner explicitement de la sémantique aux propriétés de l'élément autrement qu'au travers de son nom, ou de conventions de nommage particulières. Les modèles ainsi annotés sont plus facilement exploitables par des outils.

Dans le domaine du temps réel embarqué, comme d'ailleurs dans la plupart des domaines spécifiques, il est très compliqué, voire impossible, de définir une taxonomie complète des types utiles de propriété non fonctionnelle. De façon à être suffisamment flexible et permettre ainsi de prendre en compte des besoins non identifiés ou nouveaux, les propriétés de « NfpType » sont définies par référence tel que décrit dans le paragraphe précédent. Les propriétés de « NfpType », « valueAttrib », « unitAttrib » et « valueExpr » détaillées ci-après, sont ainsi typées par la métaclasses « Property » d'UML :

- valueAttrib est une propriété qui référence la propriété de l'élément annoté qui contiendra la valeur d'une propriété non fonctionnelle ;
- unitAttrib est une propriété qui référence la propriété de l'élément annoté qui définit la dimension (c'est-à-dire la définition des unités possibles) de la valeur de la propriété non fonctionnelle ;
- valueExpr est une propriété qui référence la propriété de l'élément qui spécifie une expression (algébrique, logique, etc.). Si une telle expression est spécifiée, cela signifie que la valeur portée par la propriété référencée par valueAttrib est calculée grâce à cette expression. Le langage VSL est une possibilité pour décrire cette expression. L'intérêt de cette propriété est de pouvoir explicitement stocker dans le modèle la manière de calculer une valeur dérivée. A noter que l'expression doit spécifier un type de retour conforme au type de la propriété référencée par valueAttrib.

Pour compléter son support aux propriétés non fonctionnelles, MARTE définit dans une annexe normative une bibliothèque de modèles contenant un ensemble de types

primitifs et de types structurés, un ensemble de types de propriété non fonctionnelle et un ensemble de dimensions incluant la description des unités pour chacune :

- la bibliothèque des types primitifs de MARTE, nommée « MARTE_PrimitiveTypes », reprend les types primitifs de UML en y ajoutant le type primitive « Real », et la définition des opérateurs mathématiques usuels nécessaires à leur manipulation. En complément, MARTE définit également dans sa bibliothèque « MARTE_DataTypes » des types de données structurées en plus de ceux d'UML. On peut ainsi décrire des vecteurs, des intervalles et des matrices de types primitifs (par exemple, des vecteurs d'entiers ou des matrices de réels) ;
- la bibliothèque « MeasurementUnits » prédéfinit un ensemble de dimensions et leurs unités respectives. On y trouve par exemple la définition des unités relatives à la dimension du temps (figure 7.2) ou encore celles relatives à la taille des données informatiques (bit, Byte, KB, etc.) ;
- la bibliothèque « BasicNFP_Types » prédéfinit des types de propriétés non fonctionnelles tel que « NFP_Duration ». Ce type permet de définir des propriétés dont les valeurs expriment des durées. En particulier, les propriétés typées par « NFP_Duration » ont une valeur nominale et peuvent également avoir une valeur au pire cas (nommée « worst »), une valeur correspondant au meilleur des cas (nommée « best ») et une précision (« precision »). La précision est la même pour toutes les valeurs de la propriété.

Enfin, notons que la bibliothèque « MARTE_PrimitiveTypes » possède également un type particulier défini comme type abstrait et utilisé comme ancêtre des autres types de propriétés non fonctionnelles, le type NFP_CommonType (figure 7.4).

«nfpType» {exprAttrib=expr}
«dataType»
<i>NFP_CommonType</i>
expr: VSL_Expression
source: VSL_SourceKind
statQ: StatisticalQualifierKind
dir: DirectionKind

Figure 7.4 – Détails du type structuré MARTE : :BasicNFP_Types : :NFP_CommonType

Les propriétés de « NFP_CommonType » sont les suivantes :

- « expr » est une propriété qui spécifie l'expression utilisée le cas échéant pour calculer la valeur de la propriété. Le type de cette propriété est « VSL_Expression », ce qui par conséquent impose l'usage de VSL pour décrire les expressions de dérivation des valeurs des propriétés non fonctionnelles. Cette règle est valable pour tous les autres types définis dans la bibliothèque « MARTE : :BasicNFP_Types » puisqu'ils généralisent tous le type abstrait, « NFP_CommonType » ;

– « source » est une propriété qui indique l'origine de la valeur indiquée par la propriété. Elle peut être estimée, mesurée, calculée ou requise (respectivement « source » vaut : *est, meas, calc* ou *req*) ;

– « statQ » est une propriété qui précise la nature statistique d'une valeur mesurée. Les valeurs possibles pour cette propriété sont définies par le type énuméré « SourceKind » dont les valeurs littérales possibles sont : *max, min, mean, variance, range, percent, distrib, determ* ou *other*.

Après avoir spécifié des dimensions, des unités, des propriétés non fonctionnelles et des types de propriétés non fonctionnelles, le sous profil « NFP » propose le concept de « NfpConstraint » pour décrire des contraintes portant sur des propriétés non fonctionnelles. On distingue trois types de contraintes au travers de la propriété « kind » dont les valeurs possibles sont les suivantes :

- *required* qui indique que la contrainte spécifie un niveau minimal requis ;
- *offered* qui indique que la contrainte spécifie un espace de valeurs supportées par les éléments contraints ;
- *contract* qui indique que la contrainte sous-jacente spécifie une relation entre des valeurs de propriétés non fonctionnelles requises et offertes.

Une contrainte possède un contexte qui identifie l'élément du modèle déterminant son contexte d'évaluation, et une liste ordonnée de références vers des éléments de modèle identifiés par l'expression booléenne la spécifiant.

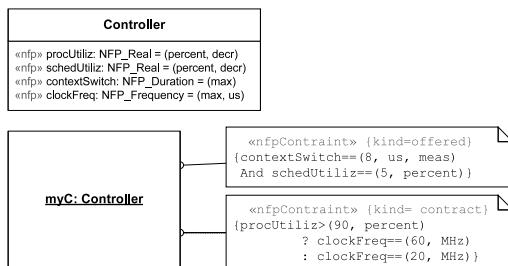


Figure 7.5 – Exemples de contraintes temps réel

La figure 7.5 montre deux exemples de contraintes s'appliquant à des propriétés non fonctionnelles. En haut de la figure se trouve la classe nommée « Controller » qui possède quatre propriétés non fonctionnelles : « procUtiliz », « schedUtiliz », « contextSwitch » et « clockFreq ». En bas à gauche est illustrée une instance de cette classe « Controller ». Cette instance est nommée « myC ». Les deux contraintes non fonctionnelles situées en bas à droite de la figure portent sur cette instance myC. La première contrainte est une propriété de « myC » : son temps de changement de contexte est de

huit microsecondes (valeur mesurée) et sa charge est de 5 %. La seconde contrainte est de type contractuelle : la charge d'utilisation du processeur est strictement supérieure à 90 % si la fréquence de son horloge est de 60 MHz, et elle est inférieure ou égale à 90 % si la fréquence de son horloge est de 20 MHz. Notons que les deux contraintes sont ici exprimées en VSL.

7.3.2. Un modèle de composants pour le temps réel embarqué

L'objectif de MARTE sur les approches à base de composant n'est pas de redéfinir un nouveau modèle de composants qui sont déjà nombreux. Au contraire, l'idée est de réutiliser au maximum l'existant et de permettre de déployer au mieux un modèle d'architecture MARTE vers un maximum de plates-formes possibles, comme AUTOSAR ou AADL. C'est donc naturellement qu'en tant que profil UML, MARTE s'est appuyé sur le modèle de composant de UML2.

Le concept de composants dans UML est disponible sous deux formes définies dans les clauses 9 et 10, respectivement intitulées « Components » et « Composites Structures ». La clause 9 étend le concept de classe de UML pour lui permettre d'avoir une structure interne, éventuellement hiérarchique, ainsi que des ports. Une telle classe, souvent appelée « classe composite », peut être reliée à ses parties internes par des « connecteurs de délégation » reliant les ports de la classe aux ports de ses parties internes. Les parties internes d'une classe composite peuvent également être assemblées par des connecteurs, alors nommés « connecteurs d'assemblage ». Les connecteurs, qu'ils soient de délégation ou d'assemblage, spécifient uniquement les chemins de communication possibles entre les composants constituant un système ; les connecteurs ne spécifient pas la nature de la communication, comme par exemple s'il s'agit d'une communication synchrone ou asynchrone.

Historiquement, UML est un langage orienté objet. De ce fait, le principe de communication privilégié entre les composants d'un système est le classique appel d'opérations, principalement synchrone ou asynchrone, et s'inscrivant dans un modèle général souvent appelé « client-serveur ». UML permet également une communication dite « par signal ». Un signal peut transporter des données et l'envoi d'un signal peut se faire en mode diffusion (*broadcast*) ou émission multipoints (*multicast*). L'ensemble des concepts définis dans la clause 10 de la norme UML permet ainsi de décrire complètement un modèle d'architecture.

La clause 9 s'appuie par extension sur les concepts de la clause 10 et complète UML en termes de description architecturale, de façon à favoriser la réutilisation des composants. Les concepts définis dans cette clause permettent de voir un composant comme une boîte noire reliée à un certain nombre d'artefacts extérieurs. Ces derniers peuvent être par exemple du code source implantant les interfaces fournies par le composant, ou encore une documentation de son interface de programmation.

Pour MARTE, c'est naturellement les composites structures de UML qui ont été choisies comme base fondatrice en termes de modélisation à base de composant. Dans ce contexte, les extensions proposées dans le sous-profil GCM (*Generic Component Model*) ont été volontairement réduites au minimum. GCM propose ainsi des concepts permettant la communication par échange de données entre composants, et un support facilitant la modélisation des interfaces fournies et requises dans le cadre d'une communication client-serveur. De plus, un apport majeur du profil GCM est sa description précise des relations entre les aspects statiques et dynamiques de la sémantique interne des composants. Cette clarification consiste principalement à décrire de façon précise ce qui se passe au niveau d'un composant, plus précisément au niveau de ses ports, c'est-à-dire ce qui se passe quand ils reçoivent un flux d'informations (réception d'un flux de données, d'un signal ou d'un appel d'opération) vis-à-vis des comportements internes de la classe composite les possédant.

En UML, la spécification des informations pouvant transiter *via* un port d'une classe, c'est-à-dire la spécification de ses interfaces fournies et requises, est définie par le type du port. En effet, un port possède deux propriétés dédiées à la définition de ses interfaces fournies et requises, mais ces propriétés sont définies comme étant dérivées, c'est-à-dire que leurs valeurs sont calculées. Dans ce cas, l'ensemble des interfaces fournies est défini comme étant l'ensemble des interfaces réalisées par le type du port (plus l'interface elle-même si le type est lui-même une interface). L'ensemble des interfaces requises est défini comme étant l'ensemble des interfaces utilisées par le type du port. Si l'on considère l'exemple de la figure 7.6, le composant « C » possède un port « p » qui offre les interfaces « I1 » et « I2 » et requiert l'interface « I3 ». En effet, la classe « P » qui type le port « p » de « C » réalise l'interface « I1 » et utilise l'interface « I2 ».

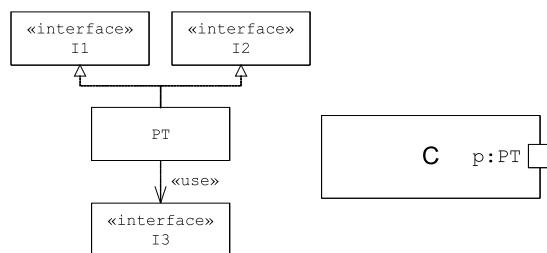


Figure 7.6 – Exemple de spécification des interfaces fournies et requises d'un composant UML

La Figure 7.7 illustre la notation spécifique proposée par UML pour visualiser graphiquement les interfaces fournies et requises d'un port.

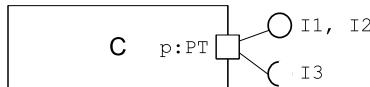


Figure 7.7 – Notation UML spécifique pour les interfaces requises et fournies

L'inconvénient principal de cette méthode de spécification des interfaces fournies et requises d'un port est qu'elle nécessite la modélisation d'une construction intermédiaire afin de définir l'élément de modèle qui va être utilisé pour typer le port. Comme illustré ci-avant, cet élément doit réaliser les interfaces que l'on veut fournir au travers d'un port, et utiliser les interfaces requises.

Ce point ayant été jugé très limitant par nombre d'utilisateurs, MARTE propose un raccourci syntaxique au travers du stéréotype « ClientServerPort » annotant les ports UML. Les propriétés « provInterface [*] » et « reqInterface [*] » du stéréotype permettent de spécifier explicitement et respectivement la liste des interfaces fournies et requises. Dans ce cas, le port ne doit pas être typé. La figure 7.8 illustre comment MARTE permet de spécifier la même chose que l'exemple de la figure 7.6 : le port « p » est maintenant stéréotypé « clientServerPort » et ses propriétés « provInterface » et « reqInterface » ont été renseignées de façon à spécifier qu'il propose les interfaces « I1 » et « I2 », et qu'il requière l'interface « I3 ».

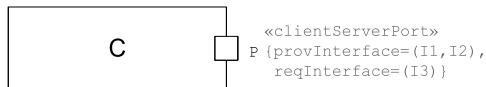


Figure 7.8 – Exemple de port client-serveur basé sur les interfaces de MARTE

En complément, MARTE propose un autre moyen de définir les ports au travers d'une unique interface, spécifiant directement à la fois les services offerts et les services requis par le port. Dans ce cas, le port est également stéréotypé « ClientServerPort ». Ce stéréotype possède une propriété particulière nommée *featureSpec* qui permet de référencer une interface particulière. Cette interface doit être stéréotypée « ClientServerSpecification ». Une interface possède alors des opérations (les services) et/ou des réceptions déclarant les signaux pouvant être perçus. Ces caractéristiques comportementales sont alors annotées avec le stéréotype « ClientServerFeature » dont la propriété « kind » détermine si elle est offerte, requise ou alors les deux à la fois. Dans ce cas, les valeurs respectives de la propriété sont *provided*, *required* ou *proreq*. La figure 7.9 montre un exemple d'utilisation de ce cette situation : le port « p » est spécifié par l'interface « I1 » qui définit que celui offre le service « service1 » et requière le service « service2 ».

En plus du modèle de communication client-serveur précédemment présenté, MARTE permet de décrire des architectures par composants communiquant par échanges de données. Ce type de modèle est possible en UML mais uniquement au travers de ses concepts d'activité et d'action, c'est-à-dire uniquement au niveau de la description comportementale d'un système. Pour compléter le modèle de composants de UML, et à l'instar de SysML et AUTOSAR, MARTE étend le concept de port de UML en introduisant le concept de port de données au travers du stéréotype « FlowPort ». Tout comme pour les ports client-serveur, la spécification de l'information transitant au travers d'un port de données se fait au travers de son type. De plus, les ports de données peuvent être atomiques ou non. Dans le premier cas, leur type est soit un signal UML, soit un type primitif, soit un type de données structurées. Dans le second cas, le port est typé par une interface UML, elle-même stéréotypée par « FlowSpecification ».

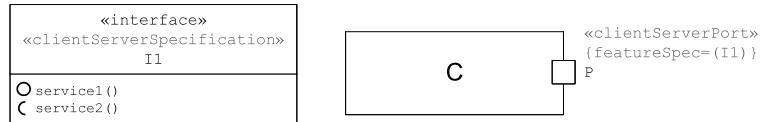


Figure 7.9 – Exemple de port défini par une interface « clientServerSpecification » de MARTE

Les propriétés d'un port de données, c'est-à-dire un port UML stéréotypé « FlowPort », sont donc un booléen nommé « *isAtomic* », dont on vient de parler, et la propriété nommée « *direction* » dont les valeurs possibles sont définies par l'énuméré « *FlowDirectionKind* » (*in*, *out* ou *inout*). Si une valeur est fixée pour cette propriété, alors le sens de toutes les données transitant par le port et spécifiées par son type doit être conforme à la valeur de la propriété.

Le modèle de composants de MARTE autorise les deux formes communément rencontrées pour la communication par données, la forme active et la forme passive³. La première forme est dite active car la réception de la donnée déclenche l'exécution d'un comportement associé au composant récepteur. Dans ce cas, la donnée reçue n'est pas stockée par le composant mais transmise à l'exécution du comportement déclenché. Cette première forme repose essentiellement sur le modèle de causalité évènementiel de UML. Dans le second cas, la forme passive, la réception de la donnée ne déclenche aucune activité au sein du composant. La donnée est stockée dans une structure *ad hoc* du composant et elle est éventuellement lue ultérieurement au cours de l'exécution d'un comportement du composant déclenchée par un autre moyen (par exemple un comportement dont l'exécution est périodique). Dans le cas de la forme passive, la donnée stockée n'est pas consommée, c'est-à-dire qu'elle n'est pas effacée, elle est

3. La forme active est souvent appelée en anglais *event-triggered* ou *push-based*, alors que la forme passive est appelée *time-triggered* ou *pull-based*.

uniquement lue. Par conséquent, une telle donnée peut être lue à plusieurs reprises, par une même exécution ou des exécutions différentes. Cette seconde forme de sémantique repose sur un usage particulier des propriétés et des connecteurs de UML.

7.4. Conclusion

L'objet de ce court chapitre n'est pas de faire une revue systématique des concepts définis dans la norme MARTE mais de donner aux lecteurs un niveau d'informations suffisant pour comprendre au mieux les chapitres suivants de cette partie du livre. En effet, les trois prochains chapitres vont en effet s'attacher à illustrer concrètement sur l'exemple du pacemaker des usages des concepts de MARTE.

7.5. Bibliographie

- [CEA 05] CEA, The Carroll Research Programme, 2005, www.carroll-research.org/uk/index.htm.
- [ESP 09] ESPINOZA H., CANCILA D., SELIC B., GÉRARD S., « Challenges in Combining SysML and MARTE for Model-Based Design of Embedded Systems », *Proceedings of the 5th European Conference on Model Driven Architecture – Foundations and Applications*, Enschede, The Nethderlands, 2009.
- [GéR 03] GÉRARD S., TERRIER F., *UML for real-time : which native concepts to use ?*, Kluwer Academic, 2003.
- [OMG 05a] OMG, OMG UML Profile for Schedulability, Performance, and Time, Version 1.1, 2005, OMG Document formal/2005-01-02.
- [OMG 05b] OMG, UML Profile for MARTE : Modeling and Analysis of Real-Time Embedded systems, RfP, 2005, OMG document realtime/05-02-06.
- [OMG 09a] OMG, OMG Policies and Procedures, Version 2.9, 2009, OMG Document pp/09-12-01.
- [OMG 09b] OMG, UML Profile for MARTE : Modeling and Analysis of Real-Time Embedded systems, Version 1.0, 2009, OMG document ptc/2009-11-02.

Chapitre 8

Modélisation de l'étude de cas avec MARTE

8.1. Introduction

Dans ce chapitre est présentée la modélisation de l'étude de cas du régulateur cardiaque (pacemaker) avec la notation UML-MARTE. Les principes MARTE ayant été donnés au chapitre précédent, nous les considérons comme compris par le lecteur de ce présent chapitre.

8.1.1. Hypothèses utilisées pour la modélisation

Cette étude repose sur la description générale d'un pacemaker [Bos 07], de ses principes de fonctionnement [BAR 10], ainsi que sur le chapitre 3, nous considérons ces documents comme des éléments du cahier des charges de notre étude de cas.

Nous avons aussi supposé que l'étude de cas modélisée en SYSML au chapitre 5 a permis de poser les bases d'une première spécification dite « système », spécification qui peut être reprise et détaillée pour notre modélisation MARTE.

La modélisation ne porte que sur la partie dite « embarqué temps réel » du cas d'étude. Il ne sera donc présenté que la partie dite du « générateur de pulsation » (appelé *Pulse Generator* dans la spécification système). En effet, la station de supervision et de configuration (appelée *Device Controller-Monitor* dans la spécification système), n'étant pas implantée dans le corps du patient et ayant moins de contraintes « temps réel », il ne nous a pas semblé nécessaire de la décrire dans le cadre de ce chapitre.

Chapitre rédigé par Jérôme DELATOUR et Joël CHAMPEAU.

8.1.2. Méthodologie de modélisation utilisée

MARTE n'est qu'une notation standardisée (un profil de la notation UML). Elle laisse ainsi à chacun le choix de sa propre méthodologie et de ses pratiques usuelles afin d'élaborer son propre processus.

Toutefois, le langage UML est un langage extrêmement riche de part les différents diagrammes offerts pour décrire les différents points de vue d'un système logiciel. Ces différents points de vue sont essentiels pour obtenir une bonne abstraction du système car un modèle représente une abstraction d'un système réel avec une intention précise. Nous allons donc présenter notre modélisation de l'étude de cas en explicitant les intentions que nous cherchons à couvrir.

Cet ensemble d'intentions est nécessairement organisé et constitue une méthodologie de modélisation. Cette méthodologie peut être appliquée en utilisant UML/MARTE et, dans ce cas, chaque intention de modélisation est associée à un ou plusieurs diagrammes. Les couples intention et diagrammes UML permettent de définir un processus d'utilisation du langage UML qui est indispensable à l'utilisation d'un tel langage dont le spectre d'applicabilité est aussi large.

C'est pourquoi dans cette introduction, nous présentons les différentes intentions que nous cherchons à couvrir pour décrire les différentes étapes de modélisation. Toutefois, cette démarche de modélisation ne cherche pas à être une méthodologie de référence pour tous les développements de systèmes temps réel. En outre, pour des raisons pédagogiques et de place, nous n'en présentons qu'une version simplifiée.

Dans nos étapes de modélisation, nous prenons en compte également les besoins de validation des modèles et de génération de code qui seront effectués dans les étapes suivantes du processus présentées dans cet ouvrage.

Dans cette méthodologie, nous sommes partis de la modélisation SysML et des spécifications système présentées au chapitre 5. Puis, nous avons raffiné le modèle afin d'expliquer les intentions suivantes :

- spécification des interfaces du système en boîte noire. Cette spécification logicielle est basée sur des cas d'utilisation raffinés associés avec les scénarios en prenant le système en boîte noire (le contenu du système n'est pas détaillé) ;
- modélisation de l'architecture logicielle basée sur une approche dite comportementale. La conception logicielle (dite système), qui permet de fournir une architecture sous forme de diagramme de classes structurelles permettant de capturer tous les comportements des scénarios par analyse boîte blanche du système. Cette intention peut elle-même être raffinée en deux intentions :
 - la première décrit l'architecture candidate représentant la décomposition du système en entités collaboratrices,

- la seconde est une description comportementale de chacune de ces entités internes au système.

A partir de ces modèles de conception, des analyses formelles ont été effectuées. Ces analyses sont présentées dans le chapitre 9. Puis, une conception détaillée a été effectuée et a permis une génération du code de cette étude de cas. Cette partie (conception détaillée et génération de code) est décrite dans le chapitre 10.

Ce processus de modélisation a été itératif et a nécessité plusieurs itérations de ces différentes étapes (spécification logicielle, conception logicielle, analyse formelle et conception détaillée). Une partie des transitions entre ces activités sont automatisées (vers l'analyse formelle et la génération de code). Ces automatisations reposent sur des techniques de transformations différentes et actuellement dédiées à différents ateliers de génie logiciel. Le lecteur ne devra donc pas être surpris de retrouver des illustrations UML/MARTE reposant sur des outils différents de modélisation.

8.1.3. Plan du chapitre

Le présent chapitre décrit uniquement les étapes de spécification et de conception logicielle dites système. Nous commençons donc par la spécification logicielle en boîte noire, puis poursuivons la conception architecturale basée sur une analyse boîte blanche des interactions entre objets et terminons par la description des comportements internes des entités de l'architecture.

8.2. Spécification logicielle

La spécification logicielle regroupe donc l'analyse en boîte noire du système en partant des cas d'utilisation et en détaillant les interactions entre le système et les acteurs externes au système. Cette étape permet de caractériser les interfaces du système en se basant sur les comportements attendus du système.

8.2.1. Cas d'utilisation et caractérisation des interfaces

Il existe une littérature abondante sur les méthodologies associées aux cas d'utilisation UML [COC 01]. Cependant dans cet ouvrage, nous prenons le parti de simplifier l'approche afin de se focaliser sur une approche qui servira, notamment, de point d'entrée pour les phases de validation du comportement des modèles. Bien que les cas d'utilisation se focalisent sur les exigences fonctionnelles du système, ils représentent cependant une articulation centrale pour la définition du modèle du système. En effet, ils regroupent et abstraient tous les comportements nécessaires et proposent une organisation qui servira ensuite de guide pour la réalisation des scénarios des interactions,

l'identification des entités internes et pour les phases de validation qui reposent sur la validation des exigences des modèles logiciels.

Majuscule conservée à dessein : ce sont les éléments du document d'origine que nous reprenons ici, de même par la suite. Nous distinguons le pacemaker objet du "Pacemaker" le modèle

En partant des cas d'utilisation établis au niveau de la modélisation SysML, nous allons raffiner le cas d'utilisation *Routine Followup*. Ce cas d'utilisation est en relation avec trois acteurs externes qui sont le *Physician*, le *Technician* et le *Patient*. Ce cas d'utilisation est détaillé en explicitant les cas d'utilisation contenus afin de caractériser les grandes interactions dans ce fonctionnement du système *Pacemaker*. Dans ce cas d'utilisation, les acteurs *Physician* et *Technician* sont identifiés comme ayant des objectifs précis par rapport au système et de ce fait sont des acteurs qui déclenchent des interactions vers le système. Ainsi, ils sont définis comme acteurs principaux. Le *Patient* est un acteur de second plan car il devra interagir avec le système *Pacemaker* afin que le système fournisse le service requis par les acteurs principaux.

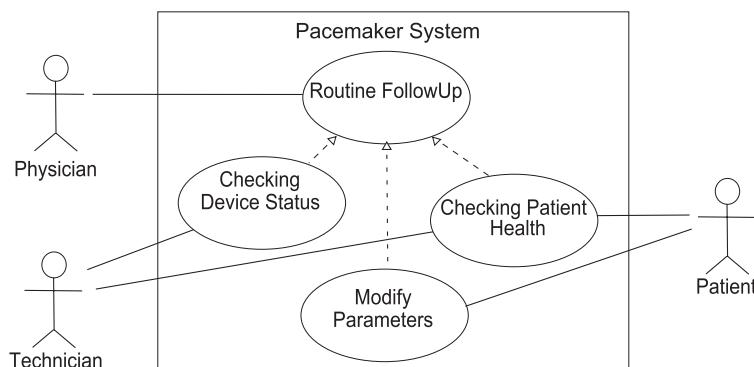
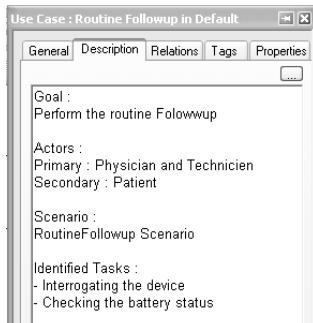


Figure 8.1 – Raffinement du cas d'utilisation *FollowUp*

Le *Physician* et le *Technician* ont comme objectif d'obtenir un état de santé du *Patient* grâce au système *Pacemaker* qui fournira les données captées du *Patient*. Pour réaliser cet objectif, le *Physician* ou le *Technician* déclenche l'interaction et le système fournit le service en ayant capté, mémorisé et traité les données venant du *Patient*.

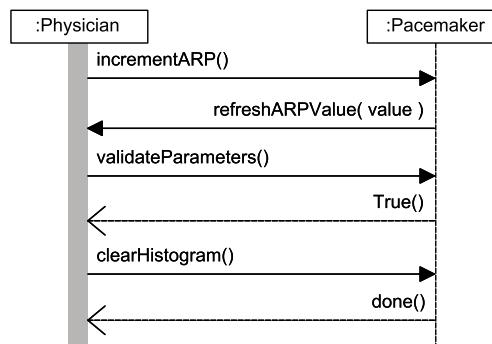
Les cas d'utilisation contiennent une description textuelle interne qui définit le contexte et les scénarios relatifs à ce cas d'utilisation. Cette description interne repose sur différents motifs, nous avons choisi de limiter la définition en intégrant l'objectif, une description textuelle, les acteurs participant à ce cas d'utilisation, l'évènement déclencheur du cas et au minimum une référence sur un scénario décrivant le comportement nominal du cas (voir figure 8.2).

En cherchant à identifier les sous-objectifs contenus dans *Routine Followup*, nous pouvons associer ces objectifs à trois cas d'utilisations *CheckingDeviceStatus*, *ModifyParameters* et *CheckingPatientHealth*. Le diagramme de la figure 8.1 présente les trois

Figure 8.2 – Description du cas d'utilisation *FollowUp*

cas d'utilisation identifiés qui précisent le cas d'utilisation général *Routine Followup*. Dans ce diagramme les relations entre les cas d'utilisation et les acteurs permettent de préciser quels sont les acteurs qui interviennent dans les scénarios associés aux cas d'utilisation.

Ce raffinement de cas d'utilisation repose sur l'utilisation de dépendances stéréotypées *Extend* car ces cas d'utilisation sont vus comme optionnels par rapport au cas d'utilisation *RoutineFollowup*. En effet, une fois le pacemaker installé le *Physician* et le *Technician* choisissent de déclencher l'interaction correspondant à chacun des cas d'utilisation. La responsabilité du *Physician* est globale pour tous les cas d'utilisation raffinés depuis celui *RoutineFollowup*. En revanche, la responsabilité du *Technician* est limité aux deux cas d'utilisation *Checking Device status* et *Checking Patient Health* ce qui se traduit par les relations entre l'acteur et les cas d'utilisation concernés.

Figure 8.3 – Scénario lié au cas d'utilisation *FollowUp*

Un scénario nominal vient décrire les interactions pour chaque cas d'utilisation avec notamment celui qui décrit le changement de la valeur ARP par incrément qui est décrit par le diagramme de séquence de la figure 8.3. Cette modification est une modification possible des paramètres du pacemaker. Les diagrammes de séquence par cas d'utilisation identifient les messages entrants et sortants du système. Cela permet ainsi de définir les interfaces du système et permet également d'identifier quels sont les messages qui doivent être traités par les objets contenus dans le système. Il est à noter que ce scénario servira de base lors de la décomposition du système en objets qui permettra d'identifier ensuite les classes du système. Les scénarios d'échecs peuvent être également ajoutés pour compléter la description des interactions en spécifiant quels sont les actions à mener lors du traitement des échecs qui peuvent survenir dans le système.

8.2.2. Domaine d'application

Comme SysML, MARTE permet de définir des types de données. A la différence de SysML, MARTE fournit par défaut une large bibliothèque de types usuels du domaine de l'embarqué, que cela soit pour définir des exigences fonctionnelles ou extra-fonctionnelles. Ainsi, nous n'avons pas à définir le type de temps (MARTE dispose, comme déjà présenté au chapitre 7, d'une sémantique riche pour décrire les contraintes temporelles et le type de temps manipulé).

«DataType» BradyCardiaModeParameter	«DataType» BradyCardiaOperatingMode	«Enumeration» BradyCardiaModeKind
+ mode: BradyCardiaOperatingMode + rate: PaceMakerFrequencyType + aLeadParameter: LeadParameter + vLeadParameter: LeadParameter + rateTolerance: NFP_Duration + waitStart: NFP_Duration + lowerRateLimit: PaceMakerFrequencyType + upperRateLimit: PaceMakerFrequencyType + modeATR: ATRParameter + modeReactive: ReactiveParameter + rateLimitTolerance: NFP_Duration + avDelay: NFP_Duration	+ pulsedChamber: ChamberKind + sensedChamber: ChamberKind + bradyCardiaMode: BradyCardiaModeKind + rateAdaptative: RatePolicyKind [0..1]	O T I D
«DataType» ATRParameter	«nfpType» «DataType» PaceMakerFrequencyType	«Enumeration» ChamberKind
	+ value: Integer + unit: PaceMakerTimeUnitKind	A V D O
«DataType» ReactiveParameter	«DataType» LeadParameter	«Enumeration» RatePolicyKind
	+ sensitivityMin: NFPTension + sensitivityMax: NFPTension + amplitudeMin: NFPTension + amplitudeMax: NFPTension + refractoryPeriod: NFP_Duration + refractoryTolerance: NFP_Duration + pulseWidthMin: NFP_Duration + pulseWidthMax: NFP_Duration	O R
«DataType» ECGEvent		«Enumeration» PaceMakerTimeUnitKind
		ppm

Figure 8.4 – Définition des types de données spécifiques au domaine

En revanche dans MARTE, il n'y a pas de description des unités du domaine (la notion de pulsation par minute ou des modes opérationnels) de la régulation cardiaque. La figure 8.4 présente ces définitions de type. Au chapitre 7, les explications concernant la définition de nouvelles unités ont déjà été données.

Concernant la définition des types médicaux pour les pulsations par minute, nous avons créé un type de données *PaceMakerFrequencyType* et une unité (*PaceMaker-TimeUnitKind*), ce afin de permettre notamment l'expression de plusieurs exigences avec cette grandeur. Toujours sur la figure 8.4, il y a la définition des paramètres des modes opérationnels, certains (*rateTolerance*, *waitStart*, *avDelay*, etc.) sont définis en utilisant le type prédéfini *NFP_Duration* pour exprimer une durée, d'autres (*lowerRateLimit*, *rate* et *upperRateLimit*) utilisent le type *PaceMakerFrequencyType* tout juste créé.

8.3. Conception logicielle système – partie architecturale

La conception logicielle système consiste à décomposer le système en entités (logicielles et/ou matérielles) qui prennent en compte, par leur coopération, l'intégralité des exigences des spécifications.

A cette étape de la modélisation, il est très souvent supposé que le futur système a des ressources infinies (en termes de mémoire, de puissance de calcul, etc.). Il s'agit principalement de s'assurer de l'obtention d'une architecture cohérente, logique et complète. Concernant les contraintes de parallélisme et de temps réel, il s'agit essentiellement d'identifier le parallélisme logique (les besoins issus des spécifications).

Les détails d'implantation (comme par exemple, la gestion des données persistantes, le démarrage et arrêt du système, la gestion des communications entre les noeuds matériels, la gestion des stimuli et réponses, etc.) sont généralement ignorés (ou simplifiés). Toutefois, si des contraintes sur ces aspects sont données par la spécification, celles-ci doivent être représentées (mais non traités en détail) dès cette étape de la conception système. Ce ne sera que dans la phase de conception détaillée que ces aspects seront explicitement traités.

De manière simplifiée, la conception système suit les grandes étapes suivantes :

- définition de l'architecture candidate, qui permet d'obtenir les premiers niveaux de décomposition du système en composants collaboratifs ;
- modélisation des comportements, qui permet de définir le comportement attendu et requis de chaque composant.

C'est un processus itératif et récursif. Pour des raisons pédagogiques, nous ne présentons que le résultat de ce processus en ne détaillant par les différentes itérations qui ont été nécessaires.

8.3.1. Architecture candidate

Nous reprenons la décomposition du système proposée par la modélisation SysML, en ne conservant que les aspects logiciels nécessaires pour notre développement.

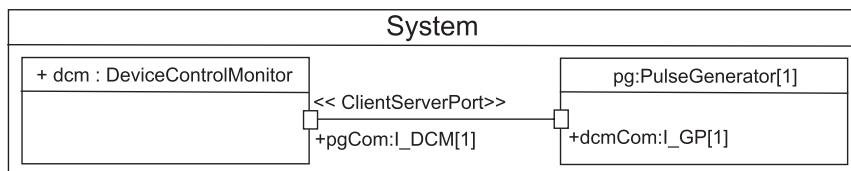


Figure 8.5 – Diagramme d’architecture générale du *Pacemaker*

Cette vue est représentée sur la figure 8.5 par un diagramme d’architecture (*Composite Structure Diagram*).

Ce diagramme ainsi que la sémantique de ses ports, ont été présentés au chapitre 11. Toutefois il est très similaire au diagramme de blocs interne de SysML (présenté au chapitre 7) et repose sur des concepts et représentations proches.

Comme précédemment indiqué, nous ne détaillerons que le générateur de pulsation (*PulseGenerator*). La modélisation SysML a par ailleurs déjà identifié une proposition de décomposition de ce composant en sous-composants (*PGController*, *Battery* et *Lead*) sans en préciser la nature (à savoir logicielle, mécanique ou matérielle).

8.3.2. Identification des composants

Pour l’établissement de l’architecture candidate et l’identification de ses composants, nous avons essentiellement utilisé la technique dite des CRC (*Class Responsibility Card*) [BEC 89]. A partir des diagrammes d’interactions associés aux cas d’utilisation écrits lors de la phase de spécification, nous identifions des composants internes au système et détaillons leurs interactions afin de s’assurer que tout message émis vers le système est correctement traité par les éléments constitutifs du système.

Cette étape est importante car elle identifie les messages entre les différents composants pour préciser les opérations allouées à chacun. Les opérations ainsi identifiées pour chacun des scénarios permettent de décrire les interfaces de chaque composant. Ainsi dans l’exemple de la figure 8.6, un scénario d’installation d’un mode opérationnel est décrit. Nous pouvons remarquer les messages *switchOn* et *startMode* qui sont ensuite reportés dans la définition des interfaces des paragraphes suivants.

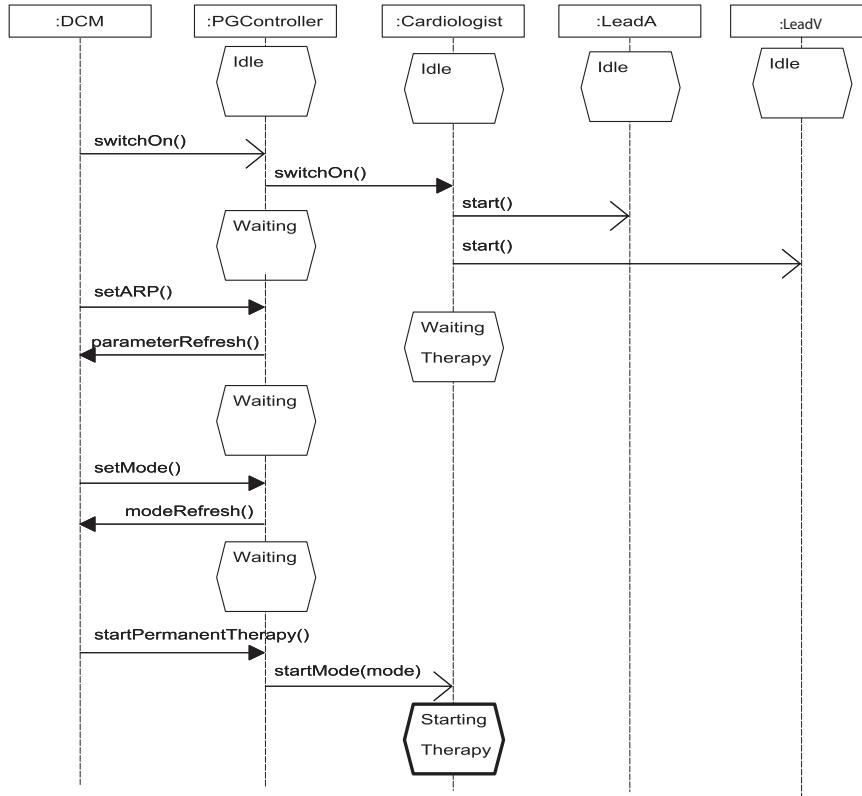


Figure 8.6 – Diagramme de séquence entre les objets du système

De plus, ces scénarios seront ensuite exploités pour modéliser le comportement interne de chaque composant. En effet, chaque message reçu par un objet doit être présenté comme déclencheur d'une transition d'un diagramme d'états et chaque message sortant d'un objet doit être émis dans le même diagramme d'états.

Pendant cette phase d'identification des interactions, nous pouvons également exprimer des contraintes vues comme des propriétés à respecter par le système. Dans ce cadre, l'utilisation de MARTE est tout à fait pertinente et nécessaire afin de fournir des contraintes non ambiguës et interprétables lors des phases de validation formelle décrites dans le chapitre suivant.

Dans la figure 8.7, nous avons une contrainte qui permet de spécifier le respect d'un temps de cycle entre deux détections de battement cardiaque envoyées par la sonde placée sur le ventricule *LeadV*. Cette contrainte est exprimée à partir de deux instants définis par $@t1$ et $@t2$. Ces instants sont identifiés comme étant les instants de

Ici, « » fait partie de la convention de nommage du standard utilisé

consommation des messages *sensedPulse()* par l'objet de type *Cardiologist*. Cette définition se fait en utilisant le stéréotype « *TimedInstantObservation* » et en spécifiant le type d'observation *obsKind* à *consume*. Cette définition lève bien l'ambiguité entre la réception du message et la consommation de celui-ci.

Au niveau de la contrainte elle-même, elle est stéréotypée « *TimedConstraint* » en précisant le type comme étant une contrainte sur une durée avec *duration*. En effet, nous exprimons ici la différence entre les instants précédemment définis @*t1* et @*t2*. Ces définitions d'instants et de contraintes sont possibles de par l'importation dans le modèle de la définition de *idealClock* définie dans MARTE comme une horloge chronométrique idéale en référence à une implantation centralisée, ce qui est notre cas ici.

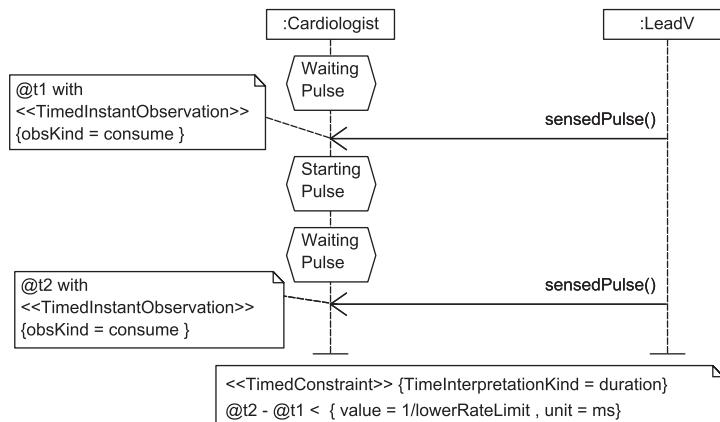


Figure 8.7 – Diagramme de séquence avec contrainte temporelle MARTE

Différents scénarios sont ainsi exprimables pour spécifier toutes les propriétés que le système doit respecter. Par la suite, ces propriétés peuvent être validées formellement en se basant sur la modélisation du comportement des entités du système.

8.3.3. Présentation de l'architecture candidate

L'application de la technique des CRC (*Class Responsibility Card*) et de bons principes de conception permettent de définir une architecture candidate identifiant les sous-composants du générateur de pulsation. Cette architecture candidate est décrite par la figure 8.8.

Les responsabilités principales des composants sont les suivantes :

- le moniteur (*controller*) est en charge de communiquer avec le moniteur-contrôleur externe (DCM), de faire appliquer les demandes du DCM et de renvoyer

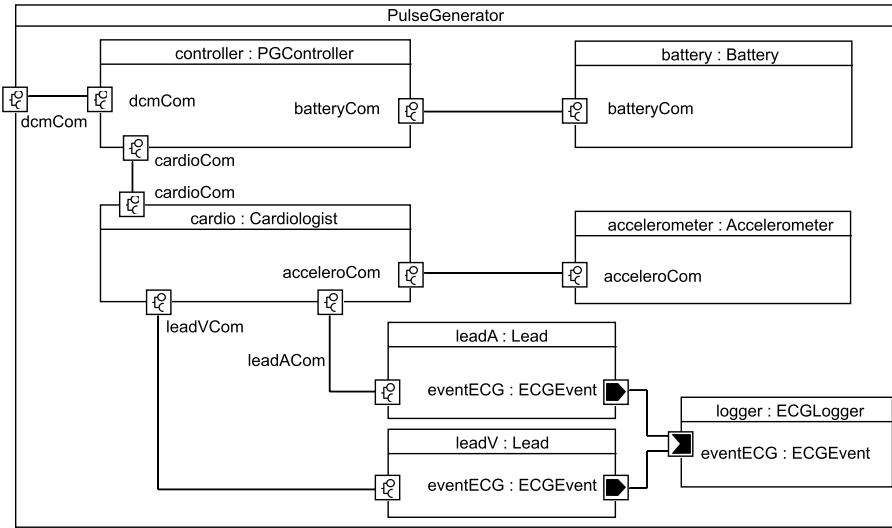


Figure 8.8 – Diagramme d'architecture du générateur de pulsation

les réponses au DCM. Il détecte (par un moyen non précisé par le cahier des charges) la proximité d'un aimant ;

– le cardiologue (*cardio*) applique une politique de surveillance de la bradycardie (voir description des modes opérationnels au chapitre 3) demandée par le moniteur. Précisons bien que ce composant est une entité de modélisation et ne désigne en aucun cas le médecin spécialisé en cardiologie qui utilisera le système. C'est une pratique courante en modélisation orientée objet que de copier la réalité. Ainsi, dans la suite de ce chapitre, quand nous parlons du cardiologue, ou des sondes, ou de la batterie, nous nous référons aux entités de modélisation ;

– les sondes (*Leads*), particularisées suivant leur positionnement sur l'oreillette (atrium, *leadA*) ou le ventricule (*leadV*) ; détectent les pulsations cardiaques et peuvent générer une stimulation cardiaque sur demande du cardiologue ;

– la batterie (*battery*) gère l'autonomie du régulateur et peut décider d'un fonctionnement en mode dégradé quand sa charge descend en-dessous de certains seuils ;

– l'enregistreur (*logger*) conserve tous les événements cardiaques que lui retransmettent les sondes (pulsations et stimulations) afin de générer un électrocardiogramme (ECG) ;

– l'accéléromètre (*accelerometer*) prévient le cardiologue dans le cas d'une détection d'activité physique plus importante du patient.

Pour des raisons de lisibilité, nous avons fait le choix de ne représenter sur la figure 8.8 que les ports et connexions principaux entre composants (par exemple, les connexions entre l'enregistreur et le moniteur n'ont pas été représentées). Les connexions entre les sondes et l'enregistreur sont des flux de données (*flowPort*). A ce stade de la conception, nous ne précisons pas encore s'il s'agit d'un flux actif ou passif. Nous modélisons juste le fait que nous gardons une trace des données émises par les sondes. En effet, l'enregistreur (*Logger*) doit conserver tous les signaux émis (stimulation cardiaque) ou détectés (battement de cœur) par les sondes afin de fournir des ECG sur demande à la station de supervision.

Les autres connexions, comme celles entre le cardiologue et le PGController, sont liées à des ports de type client-serveur (*ClientserverPort*). Ils sont donc composés d'interfaces offertes et requises. Ces aspects sont détaillés au paragraphe suivant.

A ce stade de la conception système, nous ne savons pas encore si ces composants sont logiciels ou matériels ou un mixte des deux. En effet, par exemple, pour les sondes, il est probable qu'elles seront décomposées en un sous-ensemble logiciel et matériel : une partie logicielle, s'exécutant sur une carte embarquée, et une partie électronique, placée sur le myocarde du patient et communiquant par des signaux électriques avec la carte embarquée.

8.3.4. Présentation des interfaces détaillées

Nous avons choisi de ne représenter qu'une partie des interfaces offertes et requises des différents composants, ici sur la figure 8.9, celles des interfaces offertes et requises par le composant cardiologue *Cardio*.

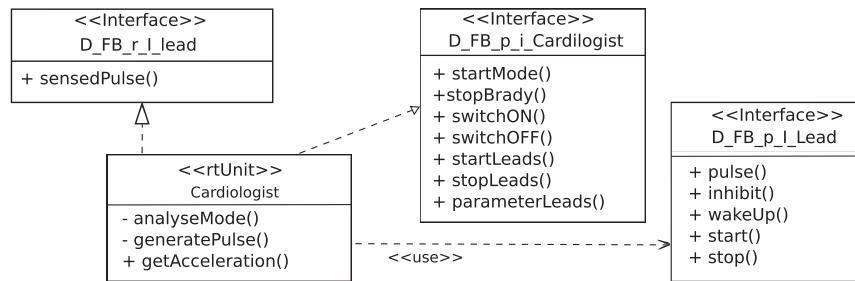


Figure 8.9 – Vue partielle des interfaces offertes et requises du cardiologue

Ces interfaces nous permettent d'identifier les communications possibles entre composants.

Nous avons une première interface offerte *D_FeatureBasedCS_r_I_Lead* par le cardiologue (le composant *Cardio*) permettant de modéliser le signalement par les sondes (*Leads*) de la détection d'un battement cardiaque (opération *SensedPulse*).

Le cardiologue utilise l'interface requise *D_FeatureBasedCS_p_I_Lead* pour communiquer avec les sondes. Il peut ainsi commander la génération d'une stimulation cardiaque (opération *pulse*), la désactivation de la sonde (opération *inhibit*) ou sa réactivation (méthode *wakeUp*). La désactivation de la sonde fait qu'elle ne viendra plus prévenir le composant *Cardiolo* de la survenue d'un battement cardiaque. En effet, suivant les modes opérationnels possibles pour la régulation, la décision de stimuler le cœur du patient peut se faire en écoutant, l'une ou l'autre des sondes. Désactiver une sonde permet ainsi au cardiologue, suivant le mode opérationnel, de n'être alerté que par la bonne sonde de la survenue d'un battement cardiaque. Les opérations *start* et *stop* modélisent la mise en marche ou l'arrêt d'une sonde (l'activation ou l'arrêt de l'envoi des signaux cardiaques à l'enregistreur).

L'autre interface offerte *D_FeatureBasedCS_p_I_Cardiologist* par le cardiologue modélise les communications possibles avec le moniteur (*controller*). Il s'agit essentiellement des commandes d'installation et de démarrage d'une politique de surveillance et d'assistance du cœur. Ces opérations seront détaillées dans la partie sur la modélisation des comportements.

8.4. Conception logicielle système – partie comportementale

L'objectif de cette phase de modélisation est de définir le comportement interne de chaque composant. Ce comportement peut être modélisé par des diagrammes d'états UML ou des diagrammes d'activité UML.

Lorsqu'un composant est dit actif (il représente un flux de contrôle indépendant de l'application), son comportement est souvent modélisé par un diagramme d'état UML. En MARTE, un composant actif est identifié par le stéréotype « *RtUnit* ». Une « *RtUnit* » *Real-time Unit* est une entité possédant son propre fil de contrôle et pouvant disposer de plusieurs modes opérationnels.

La modélisation du comportement des composants se fait par l'étude des diagrammes d'interaction (comme le diagramme de séquence donné en figure 8.7) élaborés durant l'étape précédente de définition de l'architecture candidate. Puis, il faut étendre cette description comportementale afin de s'assurer que le comportement de chaque composant recouvre tous les scénarios possibles de tous les cas d'utilisation du système.

Dans ce cas d'étude, la plupart des composants sont actifs. Nous allons décrire le comportement des principaux composants.

8.4.1. Le moniteur

Le moniteur est l'entité principale de contrôle car elle est en charge des différents mode de fonctionnement du régulateur de pulsation. Il y a cinq principaux modes de fonctionnement (représenté dans la figure 8.10) :

- le mode « *TemporaryBradyPacing* », qui permet de tester et paramétriser le régulateur. C'est un mode de fonctionnement principalement utilisé dans les phases d'implantation et d'étalonnage du pacemaker ;
- le mode « *PermanentBradying* », qui correspond au fonctionnement nominal du régulateur (en phase ambulatoire et de suivi du patient). Un mode opérationnel, adapté au patient, est appliqué ;
- le mode « *PaceNow* », qui est utilisé lors de cas d'urgence. un mode opérationnel particulier (dit « VVI ») est appliquée ;
- le mode « *PowerOnReset* », qui est déclenché lorsque la batterie du régulateur diminue et sa capacité descend sous un certain seuil. Là encore, le mode opérationnel « VVI » doit être appliqué et certaines fonctionnalités du régulateur sont alors désactivées ;
- le mode « *Magnet* », qui est un mode de test du niveau de la batterie du régulateur. En fonction du niveau de la batterie et du mode opérationnel alors engagé en fonctionnement nominal, un mode opérationnel spécifique est déclenché par le régulateur. Ce mode se déclenche par l'approche d'un aimant près de la zone du patient où est implanté le régulateur.

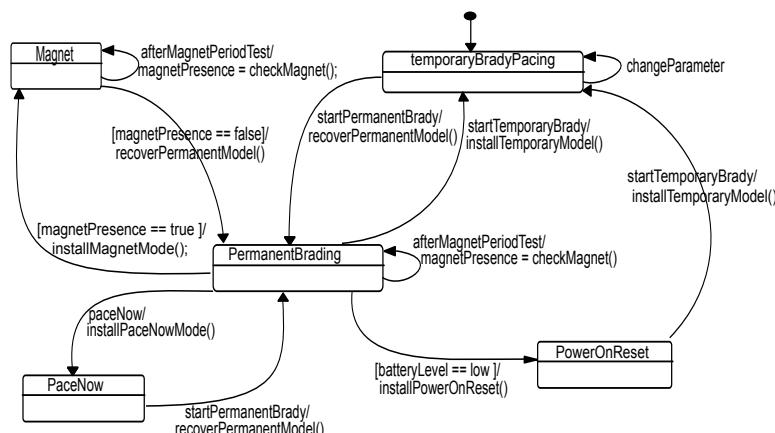


Figure 8.10 – Comportement du régulateur *Monitor* modélisé par un diagramme d'états

Pour des raisons de lisibilité, nous n'avons fait figurer que les transitions les plus importantes sur le diagramme d'états de la figure 8.10.

Dans le mode « *TemporaryBradyPacing* », nous n'avons décrit que deux transitions, l'une permet de paramétriser les différents modes opérationnels attachés aux différents modes de fonctionnement. La seconde montre le passage du mode de fonctionnement « *TemporaryBradyPacing* » au mode « *PermanentBrading* ». Cela se fait par réception du message « *startPermanentBrady* » et provoque la demande de l'exécution d'une méthode privée (« *revoverPermanentMode* ») du moniteur. A ce stade de la conception système, nous supposons que l'envoi du message *startPermanentBrady* est effectué par le moniteur-contrôleur externe (*DeviceControlMonitor*) sans donner plus de précision sur la nature de ce message (synchrone ou non). Cette hypothèse simplificatrice sera abandonnée lors de la phase de conception détaillée. Il faudra alors modéliser tous les composants nécessaires à la mise en place d'une communication sans fil entre la station de contrôle et le régulateur.

Dans le mode « *PermanentBrading* », il est testé périodiquement (*MagnetPeriodTest*) la détection de la présence d'un aimant. Si un aimant est détecté, alors le régulateur passera dans le mode « *Magnet* ».

8.4.2. Le cardiologue

Le cardiologue applique un mode opérationnel demandé par le régulateur. De ce fait, il n'a aucune connaissance des différents modes de fonctionnement du régulateur. Son comportement est représenté par la figure 8.11.

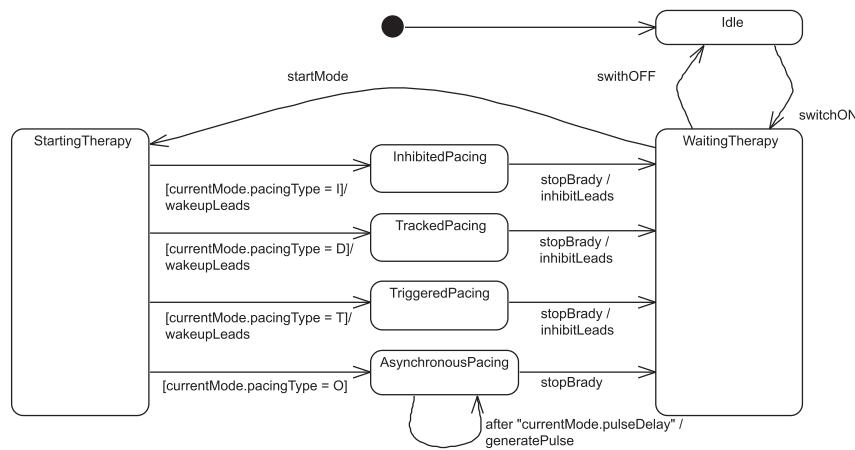


Figure 8.11 – Comportement du *Cardiologist* modélisé par un diagramme d'états

Dans son état initial (*idle*), le cardiologue est inactif. Une fois démarré (par invocation de sa méthode *switchOn*), il active les sondes (méthode *startLeads*) pour qu'elles se mettent à observer les battements cardiaques et en notifier l'enregistreur. Le cardiologue est alors dans l'état *WaitingTherapy* et attend que le régulateur lui demande de suivre un mode opérationnel donné. L'application d'un mode opérationnel (sur appel de la méthode *startMode*) se fait en deux étapes. Tout d'abord un paramétrage (exécution de la méthode *analyseMode*) est effectué suivant le mode demandé (activation ou inhibition des sondes, définition des paramètres du mode opérationnel, etc.) ; puis, en fonction du mode opérationnel demandé, il va entrer dans l'état composite correspondant. Chacun de ces états composites est décrit dans les paragraphes suivants.

8.4.3. Les modes opérationnels du cardiologue

Les modes opérationnels peuvent être distingués selon quatre grandes stratégies dites : asynchrone (*asynchronous*), déclenchée (*triggered*), inhibée (*inhibited*) ou suivie (*tracked*). Ces différents modes opérationnels sont expliqués au chapitre 3.

Mode opérationnel avec stratégie asynchrone. La plus simple des stratégies de régulation est celle dite « asynchrone » (*AsynchronousPacing*), le régulateur doit envoyer périodiquement (période *pulseDelay*) une stimulation cardiaque par l'intermédiaire de l'une ou l'autre ou des deux sondes (suivant le mode opérationnel correspondant « XXOX »). C'est la méthode privée interne *generatePulse* qui identifie quelles sondes doivent envoyées la stimulation cardiaque.

Mode opérationnel avec stratégie déclenchée. La stratégie dite « déclenchée » (*TriggeredPacing*) consiste à générer une stimulation cardiaque dès qu'un battement de cœur a été perçu par l'une des sondes. Afin de ne pas être parasité par des bruits liés à la précédente stimulation cardiaque (et le battement du cœur qui lui succède), la détection du battement de cœur ne peut se faire qu'après un délai de temps (*RefractoryPeriod*, noté RP sur le diagramme).

Il y a différents sous-modes opérationnels possibles permettant de préciser quelle sonde détecte le battement de cœur et quelle sonde génère la stimulation électrique. Toutefois, grâce au paramétrage du mode effectué avant d'entrer dans cet état composite, la description donnée par la figure 8.12 reste identique quel que soit le sous-mode opérationnel employé. En effet, l'activation des sondes (déterminée celle qui doit prévenir le cardiologue de la détection du battement de cœur) est effectué par la méthode *waheupLeads*. Et comme pour la stratégie dite « asynchrone », c'est la méthode *generatePulse* qui détermine la sonde envoyant la stimulation cardiaque. Ce constat est valable pour les modes opérationnels décrits ci-après.

Mode opérationnel avec stratégie inhibée. La stratégie dite « inhibée » (*InhibitedPacing*) est proche de celle dite « déclenchée ». Comme représenté sur la figure 8.13, si

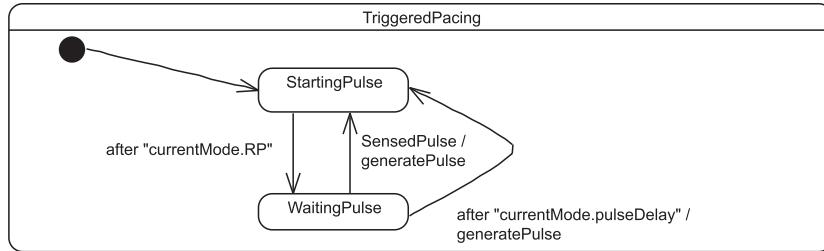


Figure 8.12 – Diagramme d’états de l’état composite *TriggeredPacing* du *Cardiologist*

le cœur se met à battre tout seul, on ne génère alors pas de stimulation cardiaque. Dans le cas contraire, on génère une stimulation cardiaque à l'aide d'une des sondes au bout d'un certain temps (*pulseDelay*).

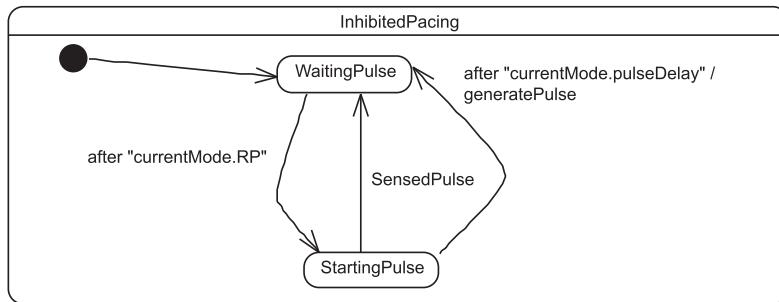


Figure 8.13 – Diagramme d’états de l’état composite *InhibitedPacing* du *Cardiologist*

Mode opérationnel avec stratégie suivie. La stratégie dite « suivie » (*TrackedPacing*) est la plus complexe (voir figure 8.14). Un battement de cœur détecté dans l’oreillette (*sensedPulsed ON LeadA*) doit être suivi d’une stimulation cardiaque dans le ventricule et ce après un délai fixe *AVDelay*, sauf si, entre temps, un battement de cœur a été détecté *sensedPulsed* dans le ventricule. Toutefois, si aucun battement de cœur n’est détecté dans l’oreillette au bout d’un certain temps (*pulseDelay*), il faut alors générer une stimulation cardiaque dans le ventricule.

8.5. Conclusion

L’objectif du chapitre était de fournir une modélisation UML pour la conception détaillée du pacemaker avec UML ainsi que son profil MARTE. Nous avons cherché à

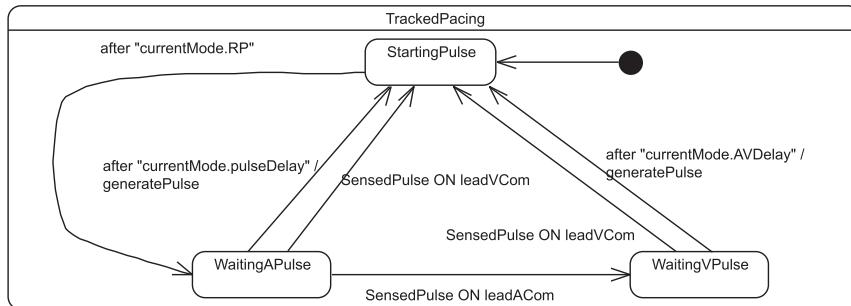


Figure 8.14 – Diagramme d’états de l’état composite *TrackedPacing* du *Cardiologist*

illustrer l’utilisation d’UML pour cette phase en partant des différents cas d’utilisation du système jusqu’à l’obtention de l’architecture interne des composants avec la description de leur comportement sous forme de *statecharts*.

Cette démarche illustre l’approche globalement orientée par les comportements utilisés avec UML puisque nous partons des interactions entre les objets pour orienter la décomposition architecturale et la description interne sous forme de machine à états. L’utilisation de MARTE a permis de spécialiser la description architecturale en utilisant notamment des ports spécifiques entre les composants mais également l’expression de contraintes temporelles dans les diagrammes d’interactions entre les composants interne de *PulseGenerator*. Ces modèles de conception système sont ensuite détaillés pour préparer la phase de génération de code pour une plate-forme spécifique mais ils sont aussi utilisés comme base pour la phase de vérification formelle décrite dans le chapitre suivant.

8.6. Bibliographie

- [BAR 10] BAROLD S., STROOBANDT R., SINNAEVE A., *Cardiac Pacemakers and Resynchronization Step by Step : An Illustrated Guide*, John Wiley & Sons, 2010.
- [BEC 89] BECK K., CUNNINGHAM W., « A laboratory for teaching object oriented thinking », *SIGPLAN Not.*, vol. 24, p. 1-6, ACM, Septembre 1989.
- [Bos 07] BOSTON SCIENTIFIC, PACEMAKER System Specification, janvier 2007.
- [COC 01] COCKBURN A., *Rédiger des cas d'utilisation efficaces*, Eyrolles, 2001.

Chapitre 9

Analyse à partir du modèle

9.1. Introduction

Les systèmes logiciels embarqués occupent aujourd’hui une place grandissante dans tous les automatismes de la vie courante. Nous les rencontrons dans les systèmes de transports : automobile (système de navigation, système de freinage, contrôleur moteur, etc.), aéronautique (navigation et contrôle du vol, pressurisation et climatisation cabine, etc.), médical (*monitoring* de patients, contrôle d’équipements médicaux tels que le pacemaker, etc.), bâtiment (sécurité incendie, contrôle ascenseur, gestion de l’alimentation électrique, etc.).

Ces logiciels assurent des fonctions automatiques et parfois critiques parce que mettant en jeu des vies humaines ou des enjeux économiques très importants. Il est donc nécessaire de garantir leur bon fonctionnement, et en premier lieu leur correction logicielle, c'est-à-dire l'absence d'erreurs ou de bogues par rapport à des spécifications fonctionnelles. De ce fait, la phase de recherche de bogues et au final de démonstration de la correction d'un système par rapport à ses spécifications a toujours occupé une part importante dans les cycles d'ingénierie des systèmes logiciels, dès les premières industrialisation de ces derniers.

Tant que ces systèmes logiciels embarqués (c'est-à-dire à la fois leur fonction et l'environnement qu'ils doivent contrôler) restaient assez simples, cette tâche de recherche de bogues pouvait être réalisée manuellement. Le corps des systèmes à valider étaient dans ce cas souvent conçu et implanté comme un logiciel séquentiel et déterministe

Chapitre rédigé par Frédéric BONIOL, Philippe DHAUSSY, Luka LE ROUX et Jean-Charles ROGER.

s'exécutant en boucle en interaction avec un environnement échantillonné. La phase de validation était faite grâce à un ensemble de tests ou de simulations. En revanche, l'élargissement progressif de la fonctionnalité des systèmes et de leur périmètre a conduit inexorablement à une augmentation drastique de leur complexité. Cette complexité est de deux types.

Premièrement, une complexité externe est liée à l'environnement que le système embarqué doit contrôler. Cet environnement est en effet de plus en plus souvent composé de plusieurs entités physiques distinctes mais dépendantes et qui doivent être contrôlées en parallèle et en cohérence. C'est le cas par exemple du système des commandes de vol d'un avion qui pilote plus de vingt gouvernes en parallèle et pour ce faire est connecté à plus de trente capteurs.

Deuxièmement, une complexité interne est liée à l'architecture logicielle du système embarqué.

Cette architecture est également de plus en plus souvent composé de plusieurs processus logiciels interagissant en parallèle.

Dans les deux cas (interne et externe), le parallélisme d'entités ou de processus conduit à une explosion du nombre de comportements possibles et par suite de scénarios de test à étudier pour valider le système. Pour se convaincre d'une telle explosion, il suffit de considérer un système composé de dix acteurs interagissant en parallèle et effectuant chacun une simple séquence de neuf actions. En supposant le cas (extrême) où ses acteurs sont totalement asynchrones (pas de synchronisation entre les actions de chaque acteur), alors le nombre d'entrelacements possibles entre les 90 actions, et donc le nombre de cas d'exécution possibles à envisager pour valider le système, est de l'ordre de 3.10^{82} soit à peu près le nombre d'atomes dans l'univers.

Il est bien sûr évident que des systèmes ou des environnements totalement asynchrones n'existent pas, les acteurs et leurs actions étant souvent partiellement synchronisés. Il n'en reste pas moins que les systèmes réels sont souvent composés de plusieurs processus (de l'ordre de dix) exécutant un grand nombre d'actions (souvent très supérieur à neuf), et que l'environnement qu'ils doivent contrôler peut également être composé de plusieurs entités (plusieurs dizaines dans le cas des commandes de vol d'un avion) produisant chacune plusieurs actions (de l'ordre de la dizaine en général). Le nombre de comportements potentiels à valider devient ainsi énorme. Ce phénomène est connu sous le terme « explosion combinatoire ». La conséquence immédiate est que les méthodes de test et de simulation manuelles ne sont plus adéquates.

Face à ce constat, plusieurs techniques ont été explorées, parmi lesquelles la famille des méthodes formelles qui ont contribué, depuis plusieurs années, à l'apport de solutions rigoureuses et puissantes pour aider les concepteurs à produire des systèmes non défaillants. Dans ce domaine et comme présenté au chapitre 2, les techniques de

model-checking [QUE 82, CLA 86] ont été fortement popularisées grâce à leur faculté d'exécuter automatiquement des preuves de propriétés sur des modèles logiciels. De nombreux outils (*model-checkers*) ont été développés dans ce but [FER 96, HOL 97, LAR 97, BER 04].

Le fonctionnement général de ces outils consiste à tenter de modéliser de façon compacte et abstraite l'ensemble des comportements possibles du système à valider et de son environnement, et de les parcourir exhaustivement et par suite de décider si l'ensemble des exécutions possibles est exempt d'erreurs. La rapidité du parcours dépend du degré de compactage de l'ensemble des comportements. Beaucoup de travaux ont été menés pour dans ce sens [VAL 91, MC. 92, GOD 96, EME 97, ALU 97, PEL 98, BOS 05, PAR 06]. Cependant, compte tenu des tailles gigantesques des ensembles considérés (souvent plusieurs ordres de grandeurs au-dessus du nombre d'atomes dans l'univers), les progrès réels des outils de vérification ne permettent pas encore de traiter des systèmes réels de taille industrielle.

Nous exposons dans ce chapitre une voie complémentaire, s'appuyant sur le *model-checking*, et qui permet de repousser plus loin les limites de l'explosion combinatoire. Cette voie cherche à traiter les deux sources de la complexité : la complexité externe (c'est-à-dire celle de l'environnement) et la complexité interne (c'est-à-dire celle du système). Plus précisément, cette approche cherche à réduire l'espace des comportements possibles en considérant un modèle explicite de l'environnement du système. En d'autres termes, il s'agit de « fermer » le système avec l'ensemble des cas d'utilisation de l'environnement auxquels il est sensé répondre, et uniquement ceux-ci. La réduction est basée sur une description formelle de ces cas d'utilisation, nommés *contextes*, avec lesquels le système interagit.

L'objectif est ainsi de guider le *model-checker* à concentrer ses efforts non plus sur l'exploration de l'espace complet des comportements, qui peut être gigantesque, mais sur une restriction pertinente de ce dernier pour la vérification de propriétés spécifiques. La vérification de propriétés devient alors possible sur l'espace d'états ainsi réduit. Cette méthode se base ainsi sur la connaissance qu'ont les concepteurs du système et leur expertise qui permet de spécifier de manière explicite l'environnement du système.

Comme toute méthode formelle, l'approche présentée repose sur un ensemble de langages formels, des passerelles d'un langage à un autre, et des outils d'exploration et de vérification. Nous développons les principes de cette approche dans la suite en utilisant deux langages particuliers : Fiacre [FAR 08], pour la description du modèle du système à valider, et CDL (*Context Description Language*), pour la description des cas d'utilisation de l'environnement considéré pour la validation.

Nous utilisons ensuite, couplés avec ces deux formalismes, trois outils. Le premier est le *model-checker* TINA SELT¹ [BER 04]. C'est un *model-checker* particulièrement bien adapté à la vérification d'exigences exprimant des invariants et formalisées en logique temporelle.

Le second est OBP Explorer² qui est un explorateur de modèle Fiacre couplé à un analyseur d'accessibilité. OBP Explorer est plus adapté à la vérification de propriétés d'accessibilité qui s'expriment à l'aide d'observateurs.

Ces deux outils sont connectés en amont à un troisième outil, OBP³, qui met en œuvre la méthode de réduction de l'espace des comportements présentée dans ce chapitre.

Cette méthode et ces outils sont illustrés sur l'étude de cas du PaceMaker décrite en UML MARTE présenté dans le chapitre 8. Le modèle de l'étude de cas est converti en Fiacre, le langage d'entrée des deux explorateurs/*model-checkers*, puis les performances des deux outils sont étudiées sur deux types d'exigences : un invariant exprimé en logique temporelle, et un observateur exprimé dans le langage CDL.

Nous montrons alors que les deux outils TINA SELT et OBP Explorer, s'ils sont utilisés sans la méthode de réduction des contextes, ne parviennent pas à dépasser le phénomène de l'explosion combinatoire inhérente à l'étude de cas du pacemaker.

Nous montrons ensuite que la prise en compte des cas d'utilisation de l'environnement du pacemaker, modélisés formellement en CDL, permet de réduire considérablement cette explosion combinatoire, avec TINA SELT comme OBP Explorer. Plus précisément, nous montrons que le partitionnement de l'environnement, opéré par OBP, en plusieurs cas d'utilisation séparés mais couvrant complètement l'ensemble des comportements de l'environnement (c'est-à-dire formant une partition au sens mathématique) permet de repousser les limites de l'explosion combinatoire. Nous montrons alors que cette démarche de partitionnement de l'environnement peut être déclenchée automatiquement chaque fois que la barrière de l'explosion combinatoire est rencontrée, permettant ainsi la vérification de propriétés sur de grands modèles.

Le chapitre est organisé comme suit. Dans la section 9.2, nous décrivons la partie du modèle UML-MARTE que nous prenons en compte pour la transformation en langage Fiacre dont nous présentons succinctement la syntaxe. Nous choisissons deux exigences que nous souhaitons vérifier sur le modèle. La section 9.3 décrit le principe de la vérification des exigences avec la prise en compte de cas d'utilisation correspondant aux modes de programmation du pacemaker. Des résultats de preuves sont

1. Développé au LAAS, www.laas.fr/tina.

2. Développé à l'ENSTA Bretagne www.obpcdl.org.

3. D'un point de vue pratique, OBP intègre l'outil OBP Explorer.

montrés pour les deux outils TINA SELT et OBP Explorer. La technique de réduction par l'exploitation des contextes, avec OBP, est décrite en section 9.4 et leur implantation en langage CDL. Nous indiquons les mesures de vérification avec les deux outils, TINA et OBP Explorer. Le chapitre se termine par un bilan et une conclusion en 9.5.

9.2. Modèle à vérifier et exigences

Nous présentons dans cette section la partie du modèle de pacemaker traduite en Fiacre à partir du modèle UML-MARTE. Nous présentons également dans les grandes lignes le langage Fiacre et les principes de traduction UML vers Fiacre. Deux exigences, sur lesquelles vont porter les vérifications, sont extraites des spécifications du pacemaker et présentées.

9.2.1. Le modèle UML-MARTE à traduire en Fiacre

Le modèle UML, introduit dans le chapitre précédent, est interprété comme un assemblage de processus (voir figure 9.1) dont le comportement est décrit à l'aide de machines à états. Ces processus communiquent par des liens de type FIFO (*First In First Out*) ou par des variables partagées (registres).

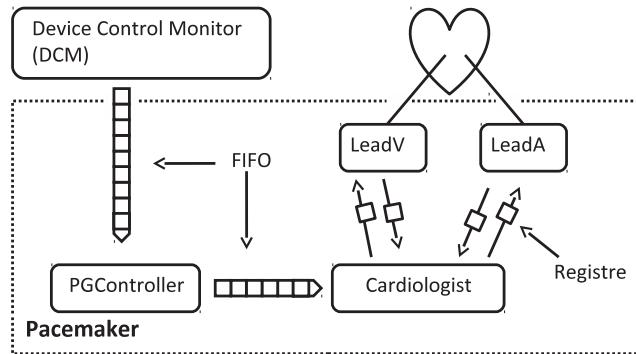


Figure 9.1 – Modèle d'architecture du pacemaker.

L'environnement du système est composé du DCM (*Device Control Monitor*) et du cœur. Le pacemaker se compose du PGController qui reçoit les consignes du DCM (modes et valeurs des paramètres), du Cardiologist qui reçoit les signaux des capteurs LeadA et de LeadV. Ceux-ci sont stimulés par le cœur. Le Cardiologist se charge en retour d'agir sur les LeadA et LeadV en fonction des modes et paramètres reçus du DCM via le PGController. Le DCM, manipulé par le médecin, envoie des messages de paramétrage au PGController. Le comportement de PGController,

LeadA et LeadV est modélisé par les automates de la figure 9.2 et celui du processus Cardiologist correspond à l'automate décrit au chapitre 8 à la figure 8.11.

Le processus PGController reçoit les messages du DCM, les traite avant de les envoier au Cardiologist. Il assure la cohérence des messages (pas de paramètres négatifs, pas de valeurs incohérentes), et envoie les paramètres au Cardiologist. Pour les appliquer au Cardiologist, il demande d'abord l'arrêt du mode en cours, envoie les nouveaux paramètres et enfin démarre le nouveau mode.

Pour appliquer les nouveaux paramètres, le processus Cardiologist doit revenir dans un mode d'arrêt nommé *Waiting*. Une fois le mode configuré, le processus Cardiologist surveille les battements (*Pulse*) du cœur et applique une politique de stimulation. Le Cardiologist est capable de compter le temps qui sépare deux battements de cœur (un cycle) et de réagir suivant les paramètres de fonctionnement.

Les Leads transforment les battements du cœur en messages compréhensibles pour le Cardiologist. Ils se chargent aussi d'appliquer la stimulation du cœur (signal *Pace*) sur ordre du Cardiologist.

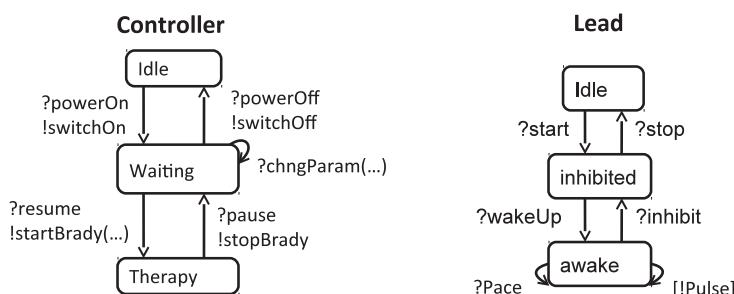


Figure 9.2 – Automates des processus PGController et Lead

Dans le modèle, nous cherchons à modéliser les comportements en tenant compte des scénarios réalistes compte tenu du comportement du DCM et du cœur comme indiqué en section 9.3.1.

9.2.2. *Le langage Fiacre*

Le langage Fiacre (Format Intermédiaire pour les Architectures de Composants Répartis Embarqués) [FAR 08] a été développé dans le cadre du projet TOPCASED comme langage pivot entre les formalismes de haut niveau comme UML, AADL, SDL et les outils d'analyse formelle. L'utilisation d'un langage formel intermédiaire apporte l'avantage de réduire le gap sémantique entre les formalismes de haut niveau et les

descriptions manipulées en interne par les outils de vérification comme les réseaux de Petri, les algèbres de processus ou les automates temporisés. Fiacre peut être considéré comme un langage disposant d'une sémantique formelle et servant de langage d'entrée à différents outils de vérification.

Fiacre est un langage formel de spécification pour décrire les aspects comportementaux et temporisés des systèmes temps réel. Il intègre les notions de processus et de composants :

- les automates (ou processus) sont décrits par un ensemble d'états, une liste de transitions entre ces états avec des constructions classiques (affectations des variables, *if-elsif-else*, *while*, compositions séquentielles), des constructions non déterministes et des communications par ports et par variables partagées ;
- les composants (*component*) décrivent les compositions de processus. Un système est construit comme une composition parallèle de composants et/ou processus qui peuvent communiquer entre eux par des ports ou variables partagées. Les priorités et les contraintes de temps sont associées aux opérations de communication.

Les processus Fiacre peuvent se synchroniser de manière synchrone avec ou sans passage de valeur par les ports. Ils peuvent également s'échanger des données au travers des files de communications asynchrones implantées par des variables partagées. C'est ce mode de communication que nous utilisons dans le modèle Fiacre. L'expression du temps dans le langage Fiacre est basée sur la sémantique des systèmes de transitions temporisés (TTS) [HEN 91]. Toute transition est associée à des contraintes de temps (temps minimum et maximum). Ces contraintes assurent que les transitions sont tirables dans des intervalles de temps définies (ni trop tôt, ni trop tard).

9.2.3. *Les principes de traduction du modèle UML en Fiacre*

Les principes de traduction du modèle UML en Fiacre sont les suivants. Les objets actifs du modèle UML sont traduits par des instances de processus Fiacre. Ceux-ci communiquent par des variables partagées. Elles permettent de modéliser des registres partagés ou des files de messages pour une communication asynchrone en mode FIFO.

Conformément aux indications du chapitre 8, les objets PGController, Cardiologist, LeadV et LeadA sont implantés par les processus Fiacre de même nom. En revanche, les objets accelerometer battery et logger n'influent pas sur le comportement des objets précédents. Ils ne sont donc pas pris en compte dans la modélisation Fiacre pour l'analyse du modèle.

Le modèle Fiacre comprend un composant *Pacemaker* (listing 1.) qui regroupe les instances des processus qui s'exécutent en parallèle (opérateur ||) : PGController, Cardiologist et deux instances du processus Lead. Nous ajoutons dans le composant une instance du processus DCM pour simuler son comportement. Ces instances

communiquent par les variables partagées de type `t_fifo` pour les files de messages ou de type `t_register` pour les registres. Ainsi, les instances des processus DCM et PGController partagent la variable `DCMToController` déclarée comme une file de messages. Les instances de PGController et Cardiologist partagent la file de messages `ControllerToCardio`. L'instance Cardiologist partage avec la première (respectivement avec la deuxième) instance de Lead les variables partagées `CardioToLeadA` et `LeadAToCardio`, (respectivement les variables `CardioToLeadV` et `LeadVToCardio`).

```
component Pacemaker is
var
DCMToController, ControllerToCardio : t_fifo,
CardioToLeadA, CardioToLeadV,
LeadAToCardio, LeadVToCardio : t_register,
par
    DCM      (& DCMToController)
    || PGController (& DCMToController, & ControllerToCardio)
    || Cardiologist (& ControllerToCardio, & LeadAToCardio,
                    & CardioToLeadA, & LeadVToCardio,
                    & CardioToLeadV)
    || Lead      (& CardioToLeadA, & LeadAToCardio)    // instance leadA
    || Lead      (& CardioToLeadV, & LeadVToCardio)    // instance leadV
end
```

Listing 1 : Déclaration Fiacre du composant *Pacemaker*

Le comportement de chaque processus est modélisé par un automate. Les états et transitions des *state-charts* UML sont traduits par des états et transitions Fiacre. Le listing 2 illustre partiellement la programmation du processus (PGController).

```
process PGController (& inputFromDCM : t_fifo_DCMToController,
                     & outputToCardio : t_fifo_ControllerToCardio) is
states
    Idle, Waiting, Therapy
var
parameters : t_parameters,    // An array of parameters (DCM level)
modification : t_modification, // A parameter index and its new value
mode : t_Mode                // An array of parameters (Cardio level)
init
    to Idle

from Idle
if not (empty inputFromDCM) then
    case first inputFromDCM of
        PowerOn ->
            inputFromDCM := dequeue inputFromDCM;
            outputToCardio := enqueue (outputToCardio, SwitchOn);
            to Waiting
            | any -> null // Should not happen
        end case
    end if;
```

```

loop
from Waiting
...
from Therapy
...

```

Listing 2 : Déclaration Fiacre du processus PGController

Pour les communications asynchrones, les opérations *first*, *dequeue* et *enqueue* permettent respectivement de lire un message en tête d'une file, de supprimer un message en tête de file et d'écrire un message en queue de file. Chaque processus se voit attribuer une file d'entrée unique. L'écriture d'un message dans cette file (*enqueue*) correspond à un envoi.

Pour cette traduction, le modèle UML est interprété avec la sémantique suivante : l'ordonnancement des processus concurrents (c'est-à-dire s'exécutant en parallèle) repose sur l'entrelacement atomique et non déterministe des transitions de chaque processus (atomique signifiant qu'une transition ne peut pas être suspendue au milieu de son exécution). En d'autres termes, pour tous les processus possédant des transitions tirables, une transition est susceptible d'être choisie de manière indéterministe et exécutée de manière atomique. Ensuite une autre transition, parmi l'ensemble des transitions tirables, est à nouveau choisie. Cette politique d'ordonnancement est conforme aux hypothèses sémantiques suivies dans le chapitre 8 lors de la modélisation UML du système du pacemaker, et correspond à la sémantique des programmes Fiacre.

Comme nous l'avons vu précédemment, la représentation du temps dans le langage Fiacre est symbolique, basée sur des intervalles. Nous verrons en section 9.2.4 que nous souhaitons vérifier des exigences qui font référence à des durées entre deux signaux ou des durées de cycles. Compte tenu des exigences que nous devons vérifier, il est nécessaire de pouvoir manipuler des valeurs correspondantes à des durées entre des événements ou des actions exécutées par un processus. Nous choisissons de manipuler des variables ou compteurs qui permettent de compter le temps en le discrétilisant. L'unité de temps choisie est paramétrable.

9.2.4. Exigences

Nous nous intéressons ici à deux exigences, introduites dans le chapitre 6, que nous souhaitons vérifier sur le modèle Fiacre. Ces exigences traitent du respect des paramètres spécifiés grâce au DCM.

La première exigence (*Exigence 1*) concerne le mode suivi (VDD). Dans le comportement du pacemaker, un signal auriculaire est suivi par un signal ventriculaire. Le

paramètre *Atrial Ventricular Delay* (*AVDelay*) définit l'intervalle de temps maximal entre ces deux évènements. En l'absence de la réponse attendue du ventricule, le pacemaker doit le stimuler. Après réception d'un signal auriculaire en mode suivi (VDD), le pacemaker ne doit donc pas « attendre » plus de *AVDelay* unités de temps. Dans le modèle, nous choisissons les millisecondes comme unités de temps. L'exigence s'exprime ainsi :

EXIGENCE 1 En mode VDD, le temps entre un signal auriculaire et une réponse ou un stimulus du ventricule ne peut dépasser la valeur du paramètre *AVDelay*.

La deuxième exigence (*Exigence 2*) décrit le temps maximum d'un cycle de battement de cœur à prendre en compte. Un paramètre *LowerRateLimit* (*LRL*) indique la période minimale du pouls en deçà de laquelle le cœur doit être stimulé par le pacemaker. Cela signifie qu'un cycle ne doit pas durer au-delà de la période indiquée par le paramètre *LRL*. Elle s'exprime ainsi :

EXIGENCE 2 La période d'un cycle ne doit pas être supérieure à la durée *LRL*.

9.3. *Model-checking* des exigences

Pour établir la vérification d'un ensemble d'exigences sur un modèle à valider, il faut disposer d'un modèle simulable et des exigences formalisées sous la forme, par exemple, de formules logiques (LTL, CTL) ou d'automates observateurs (voir chapitre 2). Il faut également modéliser les comportements de l'environnement qui interagit avec le modèle à valider. Cet environnement correspond aux différents cas d'utilisation du système que nous devons considérer pour la vérification des exigences.

9.3.1. Cas d'utilisation

Pour l'illustration de l'analyse faite sur le modèle, nous nous intéressons, dans ce chapitre, à l'analyse d'exigences pertinentes lorsque le pacemaker est en mode nominal, c'est-à-dire ambulatoire. Nous considérons ici ce contexte particulier d'utilisation. Compte tenu des spécifications fournies, les paramètres pertinents à prendre en compte sont : le choix parmi les sept modes (*mode*) et les différentes valeurs *ARP*, *VRP*, *AVDelay* et *LRL*.

Le DCM interagit avec le pacemaker en lui envoyant des valeurs pour ces différents paramètres, leurs valeurs potentielles pouvant être nombreuses. Dans ce contexte, le DCM envoie les valeurs pour les cinq paramètres : *mode*, *ARP*, *VRP*, *AVDelay* et *LRL* dans un ordre quelconque. Dans notre modélisation, l'envoi des valeurs par le DCM au pacemaker s'effectue par la communication d'un message *chgParam* à destination du processus *PGController*. Ce message a deux arguments : le nom du paramètre

à changer et la nouvelle valeur. Par exemple, pour mettre le pacemaker dans le mode *VOO*, le message émis est : *chgParam (mode, VOO)*. La figure 9.3 illustre une interaction mettant en œuvre les cinq messages pour les paramètres mentionnés.

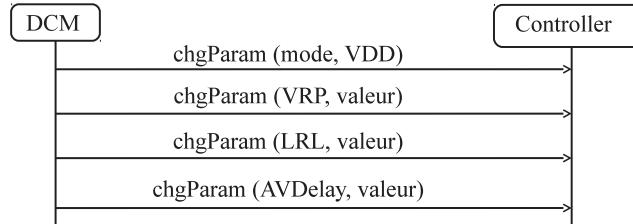


Figure 9.3 – Un exemple de scénario d’interaction (en mode VDD) entre DCM et PGController.

Les différentes valeurs que peuvent prendre ces paramètres et que nous avons pris en compte dans notre modélisation Fiacre sont :

- paramètre mode : sept modes possibles : *VOO*, *AOO*, *VVI*, *AAI*, *VVT*, *AAT*, *VDD* ;
- paramètre ARP : uniquement dans les modes *AAI* et *AAT* avec 36 valeurs possibles dans chacun d’eux (de 150 à 500 ms avec un pas de 10 ms) ;
- paramètre VRP : uniquement dans les modes *VVI*, *VVT* et *VDD* avec 36 valeurs possibles dans chacun d’eux (de 150 à 500 ms avec un pas de 10 ms) ;
- paramètre AVDelay : uniquement dans le mode *VDD* avec 24 valeurs possibles (de 70 à 300 ms avec un pas de 10 ms) ;
- paramètre LRL : dans tous les modes, sauf *AOO* et *VOO*, avec 62 valeurs possibles (de 30 à 175 ppm, avec un pas de 5 ppm jusqu’à 50 ppm, puis un pas 1 ppm jusqu’à 90 ppm, puis un pas 5 ppm jusqu’à 175 ppm).

Le nombre de valeurs que peuvent prendre les paramètres dépendent donc des modes. Il est indiqué dans le tableau 9.1. Le nombre total de combinaisons possibles, c’est-à-dire d’instances possibles du scénario présenté en figure 9.3, est ainsi de 62 498.

Modes	<i>AOO</i>	<i>VOO</i>	<i>AAI</i>	<i>AAT</i>	<i>VVI</i>	<i>VVT</i>	<i>VDD</i>
valeurs de AVDelay	–	–	–	–	–	–	24
valeurs de ARP	–	–	36	36	–	–	–
valeurs de VRP	–	–	–	–	36	36	36
valeurs de LRL	–	–	62	62	62	62	62
Nbre de combinaisons	1	1	2 232	2 232	2 232	2 232	53 568

Tableau 9.1 – Nombre de valeurs différentes pour les paramètres

9.3.2. Propriétés

L'exigence 1 peut être exprimée sous la forme d'un invariant du système. En mode VDD, le processus *Cardiologist* entre dans l'état *TrackedPacing_WaitingVPulse* lorsqu'il reçoit un signal auriculaire et attend un signal ventriculaire. La variable *timeCount* de *Cardiologist* mesure alors en unités le temps passé dans cet état. Ceci nous permet de transformer l'expression de l'exigence en l'invariant suivant :

INVARIANT 1 L'état *TrackedPacing_WaitingVPulse* du processus *Cardiologist* implique que la valeur de *timeCount* est inférieure ou égale à *AVDelay*.

Cet invariant s'écrit en logique linéaire SELT pour le *model-checker* TINA SELT :

```
[] ( cardio_1_sTrackedPacing__WaitingVPulse
    => {cardio_1_vtheMode.AVDelay} >= cardio_1_vtimeCount
);
```

L'exigence 2 pourrait également s'exprimer sous la forme d'un invariant sur le modèle. Mais ceci nécessiterait d'instrumenter le modèle par des variables supplémentaires qui pourraient être référencées par l'invariant. Sans ces variables, le temps passé dans le cycle courant n'est pas connu de notre système. Nous choisissons donc une alternative qui consiste à encoder l'exigence par un automate observateur (*Obs 2*) [HAL 93] illustré figure 9.4. Nous en expliquons le principe.

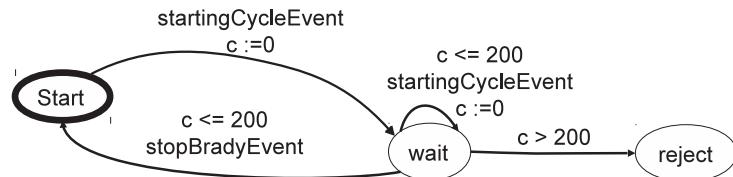


Figure 9.4 – Obs 2 : observateur temporisé encodant l'exigence 2.

Un automate observateur est un automate qui est sensible à des évènements survenant lors de l'exploration du modèle et de l'environnement : envoi et réception de messages, changement d'états des processus et des valeurs de variables. A chaque exécution d'une transition du modèle ou de l'environnement, l'observateur exécute une transition (si elle existe) correspondante à l'évènement survenu. L'observateur possède un état d'erreur *reject*. L'accès à l'état *reject* signifie que la propriété encodée par l'automate est falsifiée. Une analyse d'accessibilité consiste alors en la recherche de scénarios observés lors de l'exploration du modèle du système et de son environnement conduisant à l'état *reject* de l'observateur. Plusieurs observateurs peuvent être ainsi définis pour une même exploration du modèle. Avec ceux-ci, nous pouvons ainsi

encoder des propriétés de type sûreté et vivacité bornée. L'intérêt du codage par observateurs est de pouvoir exprimer des propriétés plus difficilement exprimables par des logiques temporelles telles que LTL ou CTL. Les observateurs n'étant pas gérés dans la version actuelle de TINA SELT, nous utilisons l'outil OBP Explorer pour leur vérification.

Concernant l'exigence 2, pour être capable de mesurer le temps séparant deux débuts de cycle, nous temporisons l'observateur. La temporisation s'effectue à l'aide d'une horloge dont la valeur s'incrémente de manière synchrone avec le temps de simulation du modèle. L'horloge peut être remise à zéro lors de l'exécution d'une transition. Les horloges des automates observateurs sont gérées au sein de l'outil OBP Explorer conformément à la sémantique des TTS (pour *Timed Transition Systems*).

Dans l'exemple figure 9.4 qui encode l'exigence 2, l'observateur passe à l'état *wait* dès l'occurrence de l'événement *startingCycleEvent*. Cet événement est défini à partir d'un prédictat *startingCycle* que nous spécifions de la manière suivante :

```
event startingCycleEvent is { startingCycle becomes true}
```

Cette spécification correspond au format de description des prédictats que nous utilisons dans le langage CDL, décrit en section 9.4.3. Le prédictat *startingCycle* est ainsi défini à partir des états *TrackedPacing_StartingPulse*, *AsynchronousPacing*, *InhibitedPacing_StartingPulse*, *TriggeredPacing_WaitingPulse* de l'automate de *Cardiologist* et de la variable *timeCount* :

```
predicate startingCycle is {
  ( {cardiologist}1@TrackedPacing_StartingPulse or
    {cardiologist}1@AsynchronousPacing or
    {cardiologist}1@InhibitedPacing_StartingPulse or
    {cardiologist}1@TriggeredPacing_WaitingPulse )
  and {cardiologist}1 : timeCount = 0}
```

Une horloge *c* est associée à l'observateur. L'événement *stopBradyEvent* est détecté lors de la mise en pause de la stimulation, c'est-à-dire lors de la réception du message *StopBrady* provenant de *PGController*. Il est spécifié par :

```
event stopBradyEvent is
  { receive StopBrady from {PGController}1 to {cardiologist}1}
```

Sur détection de cet événement, l'observateur revient dans son état initial dans l'attente d'un nouveau début de cycle. Le nœud *reject* de l'observateur est atteint si la valeur de l'horloge dépasse la valeur *LRL*. L'horloge est remise à zéro à chaque détection de *startingCycleEvent*. La valeur de l'horloge *c* représente ainsi le temps passé dans un même cycle.

9.3.3. Vérification des propriétés

Pour vérifier les propriétés sur le modèle, celui-ci doit être composé avec les cas d'utilisation décrivant l'environnement du système, puis simulé et exploré par un outil de vérification. L'exploration génère un système de transitions (SdT). Celui-ci représente tous les comportements du modèle dans son environnement sous la forme d'un graphe de configurations et de transitions. Sur ce SdT, une vérification des propriétés peut être conduite, soit en appliquant, comme dans le cas de l'outil TINA SELT (figure 9.5 (a)), des algorithmes de *model-checking* sur les formules logiques, soit en appliquant, comme dans le cas de l'outil OBP Explorer (figure 9.5 (b)), une analyse d'accessibilité des noeuds *reject* des observateurs. La difficulté liée à cette technique est la production du SdT qui peut être de grande taille, dépassant la taille de la mémoire disponible (explosion combinatoire). En cas de dépassement de la capacité mémoire disponible, la vérification n'aboutit généralement pas.

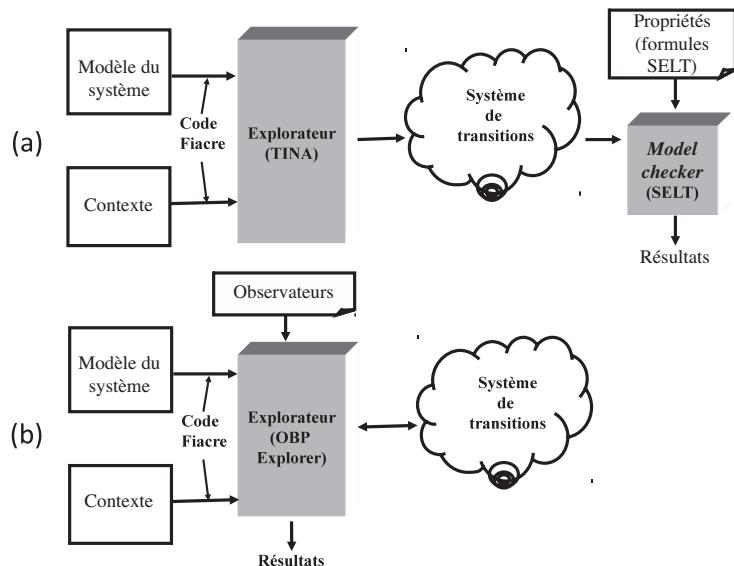


Figure 9.5 – Outils de vérification expérimentés TINA SELT et OBP Explorer

Pour pouvoir vérifier les deux propriétés mentionnées, il faut intégrer, dans le modèle Fiacre, le comportement du processus DCM pour simuler l'envoi des valeurs pour les cinq paramètres au processus PGController. Nous utilisons les deux outils mentionnés auparavant⁴ : TINA SELT pour la vérification de l'invariant 1 et l'outil OBP

4. Dans les deux cas, les vérifications sont effectuées sur une machine de type PC, possédant deux gigaoctets de mémoire.

Explorer pour la vérification de l'observateur 2. de la figure 9.4. Ces deux outils se différencient par le moment où la vérification proprement dite est faite. Dans le cas de la chaîne TINA SELT, l'explorateur TINA explore le système composé avec son environnement, génère un système de transitions (le graphe des comportements globaux), puis dans un second temps le *model-checker* SELT vérifie la propriété sur ce graphe. A l'inverse, OBP Explorer génère le graphe des comportements en prenant en compte les propriétés à vérifier (codées sous forme d'observateurs) et effectue la vérification à la volée, c'est-à-dire en même temps que la génération du graphe des comportements. La vérification, dans ce cas, se ramène simplement à la recherche des nœuds *reject* dans le graphe.

Le tableau 9.2 présente les résultats de la vérification de l'invariant 1, avec l'outil TINA SELT, pour différents nombres de combinaisons possibles de changement de valeurs pour chacun des cinq paramètres.

La vérification doit porter sur l'ensemble des combinaisons (53 568 pour le mode VDD). Or la vérification ne termine pas pour cause d'explosion combinatoire. Nous testons donc plusieurs combinaisons pour évaluer la complexité traitable avec nos outils. Nous constatons une explosion pour 2 592 combinaisons. Seulement moins de 4 % de l'espace des combinaisons, et donc de l'espace des comportements, a pu être exploré. Nous ne pouvons donc en déduire un résultat pour la vérification. Le tableau donne, pour chaque nombre de combinaisons possibles, le nombre de configurations (états du système de transitions) et de transitions explorées ainsi que le temps du *model-checking* pour ce mode *VDD*.

Nombre de combinaisons	Nombre de configurations explorées	Nombre de transitions explorées	Temps (sec)
288	321 170	748 218	12
490	541 192	1 260 682	22
768	839 834	1 956 308	33
1 134	1 233 560	2 873 322	52
1 600	1 734 954	4 041 026	219
2 592	Explosion	—	—

Tableau 9.2 – Vérification avec TINA SELT de l'invariant 1

Le tableau 9.3 présente les résultats pour la vérification de l'exigence 2, pour tous les modes, avec l'outil OBP Explorer. Tous les modes impliquent l'application d'un ensemble de 62 498 combinaisons. L'explosion combinatoire survient à partir de 352 combinaisons (soit moins de 0,6 % de l'espace des combinaisons) ce qui nous empêche également de vérifier la propriété au-delà de cette complexité.

Nous remarquons que l'explosion survient ici pour un nombre de combinaisons (352) bien inférieur au cas de la vérification de l'invariant 1 avec TINA qui était de 2 592.

Ceci s'explique par le fait que l'introduction d'un observateur, lors de l'exploration et l'analyse d'accessibilité, ajoute une composante lors de la composition avec le modèle, ce qui augmente la taille du système de transitions générée.

Nombre de combinaisons	Nombre de configurations explorées	Nombre de transitions explorées	Temps (sec)
7	40 455	48 781	2
42	299 740	364 391	10
110	744 224	909 908	25
226	1 783 438	2 238 305	64
352	Explosion	—	—

Tableau 9.3 – Vérification avec OBP Explorer de l'observateur correspondant à l'exigence 2

9.3.4. Premier bilan

Suite à ces vérifications, nous constatons que l'explosion combinatoire de la taille des SdT empêche de vérifier les exigences au-delà d'une complexité qui est vite atteinte. Le nombre de configurations atteignables du modèle exploré devient rapidement trop grand pour être contenu en mémoire car celles-ci sont générées sur l'ensemble des comportements du modèle. Et ceci malgré la prise en compte de l'environnement du système et son encodage dans un modèle Fiacre composé avec le modèle du système.

Une première explication vient du phénomène déjà observé lors de la prise en compte des observateurs lors de l'exploration du modèle du système et de son environnement par OBP Explorer. La prise en compte d'un nouvel acteur augmente la taille de l'espace d'états généré et donc renforce l'explosion combinatoire.

En généralisant cette observation, nous constatons que, si la prise en compte de l'environnement dans les modèles Fiacre est nécessaire, la prise en compte en une seule fois de tous les comportements de l'environnement ne peut conduire qu'à une explosion rapide. Pour pouvoir dépasser cette difficulté, et vérifier les propriétés sur un nombre de combinaisons plus importantes, nous allons mettre en œuvre une technique qui a pour but de limiter la complexité des SdT en décomposant l'environnement en plusieurs cas d'utilisation.

L'objectif est de vérifier les exigences, non plus lors d'une seule exploration, mais sur plusieurs explorations. Chaque exploration correspond à un sous-ensemble des comportements du DCM de manière à ce que l'exploration devienne possible sur des espaces d'états plus restreints en taille. Nous décrivons cette technique dans la section suivante.

9.4. Exploitation des contextes

Lors de la mise en œuvre classique du *model-checking*, le modèle exploré inclut les scénarios d'utilisation. Ceux-ci décrivent les interactions entre les composants de l'environnement et le modèle. Considérer le modèle avec le comportement de l'environnement amène à devoir explorer un espace d'états la plupart du temps très grand (figure 9.5).

Nous choisissons ici de spécifier l'environnement non plus sous la forme d'un seul processus global, mais comme un ensemble de scénarios explicites et séparés appelés *contextes*. Dans cette approche, l'environnement est décrit comme l'union de tous les contextes. Chaque contexte permet d'activer des sous-ensembles des comportements du modèle. Or les exigences à vérifier étant des propriétés de sûreté ou d'accessibilité, elles sont satisfaites si et seulement si elles le sont sur tous les contextes pris séparément [ROG 06]. Ce qui permet, lors du *model-checking*, de ne plus explorer un « gros » espace d'états mais plusieurs (autant que de contextes) plus petits (figure 9.6). Cette approche « diviser pour régner » permet ainsi de mener les vérifications sur des systèmes de taille importante. Nous précisons ici ce principe mis en œuvre dans l'outil OBP.

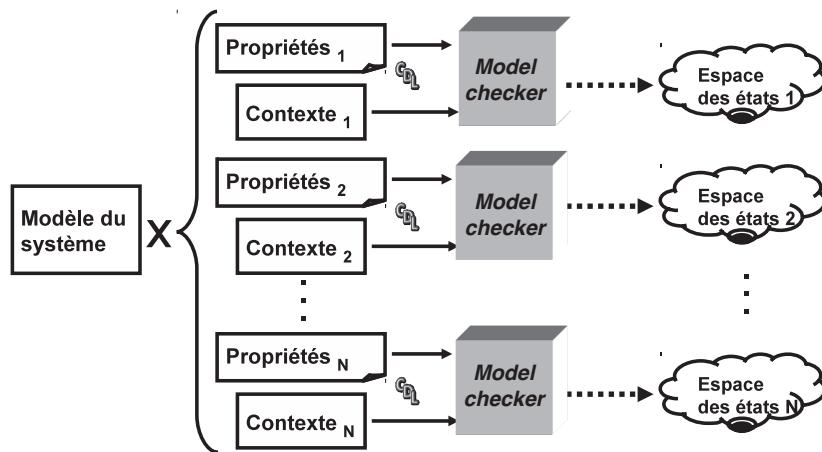


Figure 9.6 – Exploration avec identification de contextes séparés

9.4.1. Identification des scénarios de contexte

Pour identifier les contextes et les spécifier formellement, l'utilisateur se base sur la connaissance qu'il a de l'environnement du système. Ils correspondent généralement

aux modes d'utilisation du composant modélisé. Dans le contexte des systèmes embarqués réactifs, l'environnement de chaque composant d'un système est souvent bien connu. Il est donc plus efficace d'identifier cet environnement que de chercher à réduire l'espace des configurations du modèle du système à explorer.

L'objectif est de disposer de la description des sous-ensembles des comportements des acteurs de l'environnement (Contexte_i , $i \in [1..N]$ en figure 9.6) et des sous-ensembles de propriétés (Propriétés_i) associés à ces comportements. Cette identification des contextes peut déjà permettre de contourner l'explosion lors de l'exploration du modèle.

Pour que cette approche soit fondée, le processus de développement du système doit inclure une étape de spécification de l'environnement permettant d'identifier explicitement des ensembles de comportements finis mais également complets.

L'hypothèse forte que nous faisons pour mettre en œuvre ce processus méthodologique est que le concepteur est capable d'identifier toutes les interactions possibles entre le système et son environnement. Nous considérons également que chaque contexte exprimé au départ est fini, c'est-à-dire que les scénarios décrits par ce contexte ne présentent pas de comportements itératifs infinis.

Nous justifions cette hypothèse, en particulier dans le domaine de l'embarqué, par le fait que le concepteur d'un composant logiciel doit connaître précisément et complètement le périmètre (contraintes, conditions) de son utilisation pour pouvoir le développer correctement. Il serait nécessaire d'étudier formellement la validité de cette hypothèse de travail en fonction des applications ciblées. Nous n'abordons pas cet aspect qui donne lieu à un travail méthodologique spécifique à entreprendre. Néanmoins, dans l'approche actuelle, les comportements infinis de certaines entités de l'environnement doivent être inclus dans le modèle du système.

9.4.2. Partitionnement automatique des graphes de contexte

Lorsque la restriction des comportements du modèle, décrite précédemment, n'est pas suffisante, c'est-à-dire lorsque l'un des Contexte_i conduit à un espace d'états qui explose, nous mettons en œuvre un second levier de réduction de l'espace des états. Chaque contexte qui conduit à une explosion est partitionné automatiquement et récursivement en un ensemble de sous-contextes plus réduits (figure 9.7).

Pour ce faire, nous mettons en œuvre un algorithme de partitionnement récursif dans notre outil OBP. La figure 9.7 illustre la fonction `explore_mc()` pour l'exploration d'un modèle *model*, avec un contexte *context* et la vérification pour un ensemble de propriétés *pty*. Le contexte est représenté par un graphe acyclique. Ce graphe est composé avec le modèle lors de l'exploration. En cas d'explosion, ce contexte est

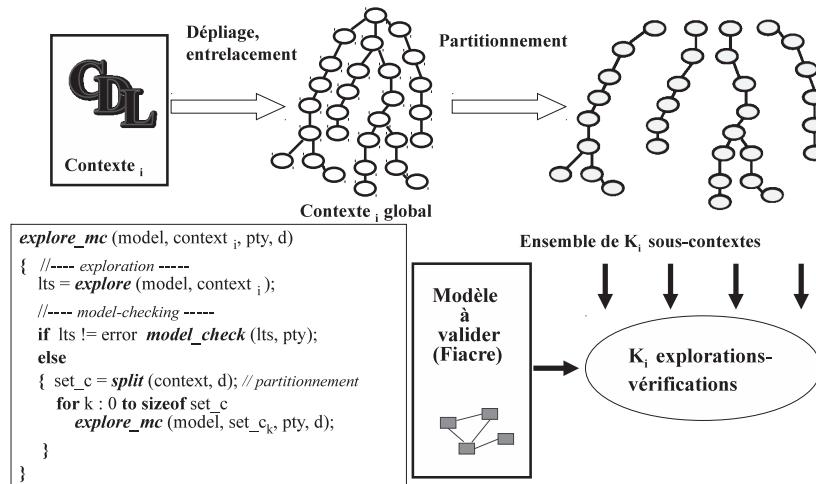


Figure 9.7 – Partitionnement d'un contexte et vérification pour chaque partition

automatiquement partitionné en plusieurs sous-graphes (avec la prise en compte d'un paramètre d , qui spécifie une profondeur du graphe à partir de laquelle la partition s'opère). Ce traitement récursif s'exécute jusqu'à ce que toutes les explorations aient été menées sans explosion.

De fait, pour chaque Contexte_i, nous transformons une vérification globale en K_i vérifications plus petites, où K_i est le nombre de sous-contextes obtenus après le partitionnement du Contexte_i. Au final, pour l'ensemble des contextes, cela revient à transformer N vérifications (autant que de contextes) en $N' = \sum_{i=1}^N K_i$ petites vérifications. Il faut noter que la technique de partitionnement mise en œuvre respecte le principe suivant : pour un contexte donné, l'union des exécutions, décrite par l'ensemble des sous-contextes générés par le partitionnement du contexte, inclut les exécutions décrites par ce contexte initial. Les propriétés sont donc préservées par le partitionnement du contexte comme démontré dans [ROG 06].

9.4.3. Le langage CDL

CDL [DHA 09] est un DSL⁵ qui s'inspire des *Use Case Chart* de [WHI 06] basé sur des diagrammes d'activités et de séquences. L'objectif de CDL est de permettre la modélisation formelle des contextes du systèmes (c'est-à-dire les Contextes_i mentionnés dans les sections précédentes). Ce formalisme permet de décrire le comportement

5. Domain Specific Language.

de plusieurs entités (nommées *acteurs*) composant l'environnement. Celles-ci s'exécutent en parallèle et interagissent avec le modèle du système.

Doté d'une syntaxe graphique et textuelle, un modèle CDL décrit, d'une part, un scénario avec des diagrammes d'activités et de séquences. D'autre part, il décrit les propriétés à vérifier en se basant sur des patrons de définition de propriétés. Un métamodèle de CDL a été défini, ainsi qu'une syntaxe et une sémantique formelles qui sont décrites [DHA 11a] en termes de traces⁶, s'inspirant des travaux de [HAU 05] et [WHI 06].

Un modèle CDL est structuré de manière hiérarchique. Un premier niveau déclare l'ensemble des acteurs constituant l'environnement du système et évoluant en parallèle les uns des autres. A ce niveau, un contexte CDL peut être représenté (de façon simplifiée) par une construction du type $A_1 \parallel A_2 \parallel \dots \parallel A_n$ où chaque A_i représente un acteur de l'environnement. Chaque A_i est ensuite détaillé sous la forme d'un diagramme d'activités, c'est-à-dire comme une composition de MSC ou de sous-diagrammes d'activités. Les compositions possibles à ce niveau étant l'enchaînement (c'est-à-dire la séquence) ou le choix (c'est-à-dire l'alternative) entre deux ou plusieurs branches. Ensuite chaque MSC est décrit comme un diagramme de séquences de type UML2.0 [ITU 96] simplifiés décrivant les interactions entre l'acteur et le système. Formellement, un modèle CDL peut être considéré comme un ensemble de MSC composés entre eux à l'aide de trois opérateurs : la séquence (*seq*), le parallèle (*par*) et l'alternative (*alt*).

Lors de la compilation d'un modèle CDL, les diagrammes correspondant à chaque acteur sont dépliés (mise à plat de chaque boucle finie) puis entrelacés entre eux conformément à la sémantique du parallélisme de Fiacre et UML-MARTE. L'entrelacement de l'ensemble des MSC, décrivant le comportement du contexte, génère un graphe représentant toutes les exécutions des acteurs de l'environnement considéré. Ce graphe est ensuite partitionné, de manière à générer un ensemble de sous-graphes correspondant aux sous-contextes, comme mentionné en 9.4.2. Lors de l'exploration par le vérificateur, chaque sous-graphe est composé (figure 9.7) avec le modèle à valider et les propriétés sont vérifiées sur le résultat de cette composition.

La figure 9.8 représente graphiquement un modèle CDL du DCM (qui est l'unique acteur dans l'environnement du pacemaker). Selon ce contexte, le DCM peut (l'opération « alternative » est représentée graphiquement par un losange) exécuter l'un des sept MSC *SetModeX*. Chaque MSC *SetModeX* modélise l'envoi d'un ordre de changement de mode du DCM vers le pacemaker. Par exemple *SetModeAOO* est un MSC contenant l'unique interaction « chgParam (mode, AOO) » de DCM vers PG-Controller (voir figure 9.3). Après exécution de ce changement de mode, le DCM

6. Pour la syntaxe détaillée, voir aussi [DHA 11b] disponible sur le site www.obpcdl.org

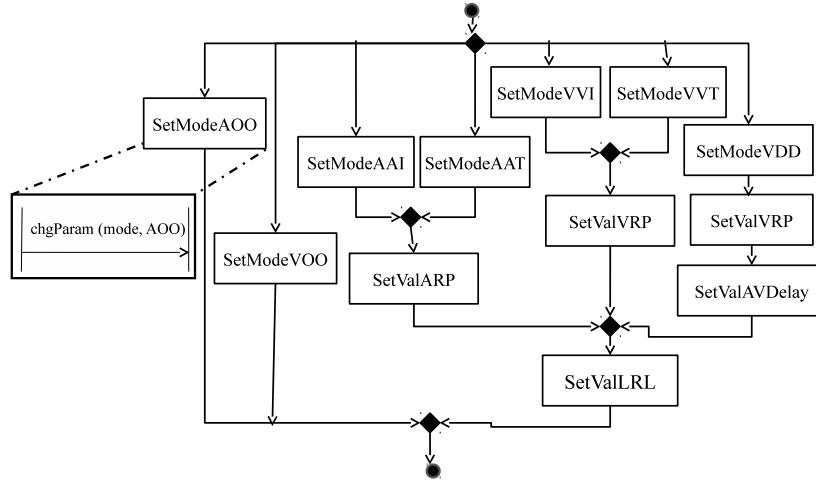


Figure 9.8 – Modèle CDL du DCM

continue par l'exécution de zéro, un, deux ou trois MSC *SetValX* selon la branche prise lors de la première alternative. Chaque *SetValX* est un MSC qui modélise toutes les alternatives possibles pour l'affectation d'une valeur au paramètre *X*, où *X* est *ARP*, *VRP*, *AVDelay* ou *LRL*. Par exemple, le MSC *setValARP* (figure 9.9) modélise une alternative entre 36 affectations possibles du paramètre *ARP*. Le modèle CDL de la figure 9.8 représente donc l'ensemble des 62 498 combinaisons possibles en entrée du pacemaker.

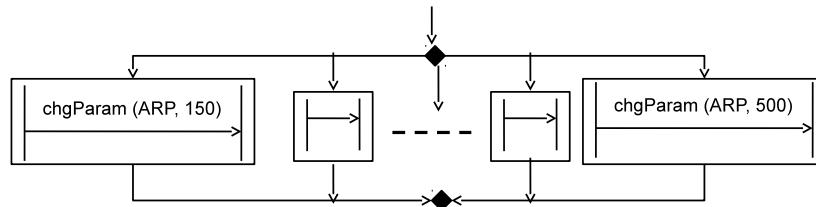


Figure 9.9 – SetValARP : échange des valeurs pour le paramètre ARP

Notons enfin que CDL permet aussi de spécifier des propriétés à partir de patrons de définition de propriétés de manière à aider l'utilisateur à les formaliser. Nous n'abor-dons pas cet aspect dans ce chapitre.

9.4.4. Exploitation des modèles CDL dans un model-checker

L'outil OBP prend en entrée les modèles CDL pour les convertir de deux façons comme illustré sur la figure 9.10. Soit OBP traduit les diagrammes CDL en programmes Fiacre pour qu'ils soient fournis à TINA, soit OBP génère des données pour l'explorateur OBP Explorer, intégré lui-même dans OBP.

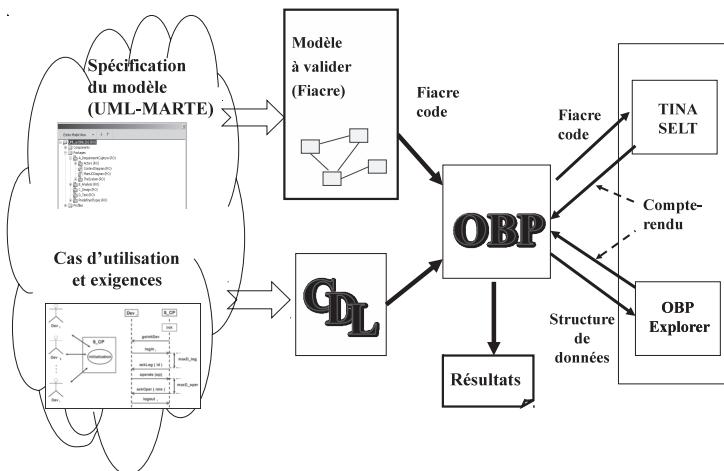


Figure 9.10 – Transformation d'un modèle CDL avec OBP

Les programmes Fiacre ou les données générées par OBP décrivent dans les deux cas un ensemble de graphes acycliques de contexte. Ceux-ci représentent l'ensemble des interactions possibles entre l'environnement et le modèle. Pour valider ce dernier, il est nécessaire et suffisant de le composer avec chaque graphe [ROG 06]. Les propriétés doivent alors être vérifiées sur le résultat de chacune des compositions. Dans le cas de Tina, les formules SELT sont vérifiées par *model-checking*. Dans le cas d'OBP Explorer, une analyse d'accès est menée sur le résultat de la composition entre un graphe généré, un ensemble d'observateurs et le modèle. Dans ces deux cas, OBP récupère les résultats provenant de SELT ou d'OBP Explorer et les forme pour les rendre compréhensibles par l'utilisateur. La partition du contexte par OBP en un ensemble de graphes permet d'aboutir, lors de la composition, à des systèmes de transitions de taille limitée rendant possible l'analyse d'accès dans le cas d'OBP Explorer ou le *model-checking* dans le cas de TINA.

9.4.5. Description d'un contexte CDL

Pour le cas d'étude traité ici, le contexte CDL décrit les interactions entre le DCM et le processus PGController. Il ne contient donc qu'un seul acteur CDL. Les interactions sont décrites sous la forme d'alternatives et de séquences qui modélisent tous les scénarios envisagés, c'est-à-dire les combinaisons de valeurs fournies au PGController. Un des scénarios, comme illustré figure 9.8, décrit l'envoi au processusPGController de la valeur du mode puis une valeur pour chacun des paramètres *ARP*, *VRP*, *LRL* et *AVDelay*. La version textuelle de l'ensemble des scénarios illustré par la figure 9.8 est illustrée listing 3. L'opérateur « [] » spécifie l'alternative et le « ; » la séquence. Le scénario *allModes* référence *refractoryPeriodProperty*. Il s'agit de la propriété correspondant à l'observateur *Obs 2* à vérifier.

```
cdl allModes is {
    properties refractoryPeriodProperty;
    {
        { set_mode_AOO [] set_mode_VOO }
        []
        {
            {
                { { set_mode_AAI [] set_mode_AAT }; set_val_ARP }
                []
                { { set_mode_VVI [] set_mode_VVT }; set_val_VRP }
                []
                { set_mode_VDD; set_val_VRP; set_val_AVDelay }
            };
            set_val_LRL
        }
    }
}
```

Listing 3 : Le modèle CDL (version textuelle) pour tous les modes.

Avec :

- *set_val_ARP* équivalent à : *setARP_150* [] ... [] *setARP_500*;
- *set_val_VRP* équivalent à : *setVRP_150* [] ... [] *setVRP_500*;
- *set_val_AVDelay* équivalent à : *setAVDelay_70* [] ... [] *setAVDelay_300*;
- *set_val_LRL* équivalent à : *setLRL_30* [] ... [] *setLRL_175*.

9.4.6. Résultats

Nous présentons les résultats de la vérification sur le pacemaker en mettant en œuvre les contextes CDL et la technique du partitionnement des graphes de contextes.

Le tableau 9.4 présente les résultats pour la vérification de l'exigence 1 pour le mode *VDD* avec l'outil TINA SELT avec l'exploitation des contextes. Contrairement à l'expérimentation montrée en section 9.3.3, la vérification se termine en un demi-heure

Nombre de combinaisons	Nombre de sous-contextes générés	Nombre (cumulé) de configurations explorées	Nombre (cumulé) de transitions explorées	Temps d'exploration (sec)
1 600	10	1 735 440	4 042 232	72
2 592	12	2 807 562	6 538 731	125
6 144	16	6 598 944	15 367 920	268
12 000	20	12 790 310	29 786 725	508
20 736	24	22 059 672	51 371 796	855
32 928	588	34 932 422	81 367 573	1 499
47 616	768	50 791 728	118 294 920	2 017
53 568	864	57 148 944	133 100 760	2 270

Tableau 9.4 – Vérification de l'invariant 1 (mode VDD) avec TINA SELT et l'exploitation des contextes.

environ y compris pour le DCM complet, c'est-à-dire offrant les 53568 combinaisons possibles du mode VDD (dernière ligne du tableau). La vérification montre que l'exigence 1. est satisfaite par le modèle du pacemaker.

Afin d'illustrer l'évolution du temps de vérification, nous avons également testé la méthode de vérification sur des sous-ensembles du DCM allant de 1 600 combinaisons possibles (configuration du contexte à partir de laquelle la méthode classique de la section 9.3.3 explose) à la totalité des combinaisons. On voit alors que le temps de vérification évolue de façon faiblement exponentielle, tout en restant raisonnable (au maximum une demi-heure) pour un espace d'états très grands (jusqu'à 57 millions de configurations).

Le tableau 9.5 présente les résultats pour la vérification de l'observateur *Obs 2* correspondant à l'exigence 2, pour tous les modes, avec l'exploitation du modèle CDL du DCM par OBP. Ici encore, on constate que la vérification termine en six heures environ y compris pour le modèle complet du DCM offrant 62 498 combinaisons possibles (dernière ligne du tableau). L'espace d'états exploré contient environ 535 millions de configurations. Comme précédemment, l'évolution du temps de vérification est montré par une succession d'expérimentations réalisées sur des contextes de plus en plus grands (contenant de 352 combinaisons à la totalité). Ici encore, l'évolution du temps est exponentielle, mais reste faisable en pratique (environ six heures).

Nous remarquons une nouvelle fois que la taille du système de transitions généré est plus importante que dans le cas de l'exploration avec TINA. Pour un même nombre de combinaisons, sa taille est ici dix fois supérieure. Comme évoqué en section 9.3.3, l'introduction de l'observateur lors de l'exploration et l'analyse d'accessibilité augmente la complexité du système de transitions généré.

Nombre de combinaisons	Nombre de sous-contextes générés	Nombre (cumulés) de configurations explorées	Nombre (cumulés) de transitions explorées	Temps d'exploration (sec)
352	7	2 680 017	3 362 387	92
578	12	4 519 385	5 729 766	238
884	13	7 033 893	8 987 094	241
1 282	14	10 179 272	13 069 412	437
3 746	72	30 869 430	40 072 535	1 686
8 194	228	67 893 326	88 639 720	3 963
15 202	344	126 267 775	16 5504 813	6 355
25 346	484	21 2075 376	27 8725 276	9 679
39 202	648	329 522 688	434 009 683	14 226
55 554	636	476 083 965	628 001 017	19 779
62 498	940	535 871 149	706 896 539	22 238

Tableau 9.5 – Vérification de l'exigence 2 avec OBP Explorer et l'exploitation des contextes.

9.5. Bilan

L'identification des contextes CDL et la technique de partitionnement permettent de restreindre l'exécution des modèles et de mener complètement les explorations. Dans le cas de TINA ou de l'analyseur interne à OBP, l'exploration de tous les comportements est impossible sans la mise en œuvre d'un modèle de contexte et de la technique de partitionnement de celui-ci. Nous avons montré que l'application de l'ensemble des 62 498 combinaisons des valeurs des paramètres associées aux modes choisis peut mener à une vérification des deux exigences considérées. De plus, l'apport des modèles CDL est d'offrir un cadre formel pour décrire les exigences pouvant être traduites automatiquement en automates observateurs.

Dans des applications industrielles, la description des contextes interagissant avec le modèle à valider est souvent informelle, parfois incomplète. L'approche permet à l'utilisateur de formaliser cet environnement et de préciser, dans un ensemble de modèles CDL, les cas d'utilisation du composant développé. Cette formalisation ne peut être que bénéfique pour une meilleure conception même si l'utilisateur ne l'exploite pas, par la suite, pour des analyses formelles. Cette formalisation est basée sur des diagrammes d'activités et des diagrammes de séquences qui sont d'un accès aisé pour un ingénieur. Conceptuellement, les principes de CDL peuvent donc être implantés dans d'autres formalismes plus standards comme les diagrammes d'activités et séquences UML.

Dans notre illustration, nous avons considéré le DCM comme le seul acteur de l'environnement. Du fait de n'avoir, dans ce cas, qu'un acteur unique, l'intérêt du partitionnement automatique peut sembler limité. Ce n'est pas le cas lorsque l'environnement

est composé de plusieurs acteurs. En effet, OBP construit à partir du comportement de chaque acteur, un graphe de l'ensemble des comportements des acteurs en considérant l'entrelacement des comportements. Ce graphe est ensuite partitionné avec la technique décrite dans ce chapitre.

Nous pourrions mener des vérifications sur des modèles plus complexes, par exemple pour un nombre de combinaisons beaucoup plus important. Le temps d'exploration et de vérification augmenterait en conséquence. Pour accélérer les vérifications, il faudrait accroître les performances de l'outillage en augmentant d'une part la mémoire pour limiter les opérations de partitionnement des contextes et, d'autre part, en parallélisant le traitement des contextes sur un réseau de machines [DHA 11a].

9.6. Conclusion

Ce chapitre avait pour objectif de présenter une technique de vérification (par *model-checking*) de propriétés sur le modèle UML-MARTE décrit au chapitre 8.

L'approche décrite cherche à réduire l'espace des comportements possibles en considérant un modèle explicite de l'environnement du système formalisé sous la forme d'une union de contextes. Ce modèle est décrit à l'aide du langage CDL qui permet de décrire les interactions de l'environnement avec le modèle à valider.

Un contexte ainsi formalisé peut être exploité par l'outil OBP et partitionné en des sous-contextes qui sont composés avec le modèle à valider. Ce partitionnement permet de réduire le nombre de comportements du modèle à explorer à chaque vérification, et ainsi de faire reculer l'explosion combinatoire. Lors des explorations, nous mettons en œuvre et vérifions des observateurs. Nous avons illustré cette technique pour un ensemble d'interactions entre le *DCM* et le contrôleur du pacemaker. Elle est mise en œuvre sur le *model-checker* TINA et l'analyseur OBP Explorer, interne à OBP.

9.7. Bibliographie

- [ALU 97] ALUR R., BRAYTON R. K., HENZINGER T. A., QADEER S., RAJAMANI S. K., « Partial-Order Reduction in Symbolic State Space Exploration », *Computer Aided Verification*, p. 340-351, 1997.
- [BER 04] BERTHOMIEU B., RIBET P., VERDANAT F., « The tool TINA - Construction of Abstract State Spaces for Petri Nets and Time Petri Nets », *International Journal of Production Research*, vol. 42, 2004.
- [BOS 05] BOSNACKI D., HOLZMANN G. J., « Improving Spin's Partial-Order Reduction for Breadth-First Search », *SPIN*, p. 91-105, 2005.
- [CLA 86] CLARKE E., EMERSON E., SISTLA A., « Automatic verification of finite-state concurrent systems using temporal logic specifications », *ACM Trans. Program. Lang. Syst.*, vol. 8, n° 2, p. 244-263, ACM, 1986.

- [DHA 09] DHAUSSY P., PILLAIN P.-Y., CREFF S., RAJI A., TRAON Y. L., BAUDRY B., « Evaluating Context Descriptions and Property Definition Patterns for Software Formal Validation », *12th IEEE/ACM conf. Model Driven Engineering Languages and Systems (Models'09)*, vol. LNCS 5795, Springer-Verlag, p. 438-452, 2009.
- [DHA 11a] DHAUSSY P., BONIOL F., ROGER J.-C., « Reducing State Explosion with Context Modeling for Model-Checking », *13th IEEE International High Assurance Systems Engineering Symposium (Hase'11)*, Boca Raton, Etats-Unis, 2011.
- [DHA 11b] DHAUSSY P., ROGER J.-C., CDL (Context Description Language) : Syntaxe et sémantique, Rapport, ENSTA-Bretagne, 2011.
- [EME 97] EMERSON E., JHA S., PELED D., « Combining Partial Order and Symmetry Reductions », *Tools and Algorithms for the Construction and Analysis of Systems*, Enschede, Pays-Bas, Springer Verlag, LNCS 1217, p. 19-34, 1997.
- [FAR 08] FARAIL P., GAUFILLET P., PERES F., BODEVEIX J.-P., FILALI M., BERTHOMIEU B., RODRIGO S., VERNADAT F., GARAVEL H., LANG F., « FLACRE : an intermediate language for model verification in the TOPCASED environment », *European Congress on Embedded Real-Time Software (ERTS)*, SEE, janvier 2008.
- [FER 96] FERNANDEZ J.-C., GARAVEL H., KERBRAT A., MOUNIER L., MATEESCU R., SIGHIREANU M., « CADP : A Protocol Validation and Verification Toolbox », *CAV '96 : Proceedings of the 8th International Conference on Computer Aided Verification*, London, UK, Springer-Verlag, p. 437-440, 1996.
- [GOD 96] GODEFROID P., PELED D., STASKAUSKAS M. G., « Using Partial-Order Methods in the Formal Validation of Industrial Concurrent Programs », *International Symposium on Software Testing and Analysis*, p. 261-269, 1996.
- [HAL 93] HALBWACHS N., LAGNIER F., RAYMOND P., « Synchronous observers and the verification of reactive systems », NIVAT M., RATTRAY C., RUS T., SCOLLO G., Eds., *Third Int. Conf. on Algebraic Methodology and Software Technology, AMAST'93*, Twente, Workshops in Computing, Springer Verlag, June 1993.
- [HAU 05] HAUGEN O., HUSA K. E., RUNDE R. K., STOLEN K., « STAIRS towards formal design with sequence diagrams. », *Software and System Modeling*, vol. 4, n° 4, p. 355-357, 2005.
- [HEN 91] HENZINGER T., MANNA Z., PNUELI A., « Timed Transition Systems », *Proceedings of the 1991 REX Workshop*, 1991.
- [HOL 97] HOLZMANN G., « The Model Checker SPIN », *Software Engineering*, vol. 23, n° 5, p. 279-295, 1997.
- [ITU 96] ITU, « Message Sequence Chart (MSC) », *ITU-T Recommendation Z.120*, Geneva, 1996.
- [LAR 97] LARSEN K. G., PETTERSSON P., YI W., « UPPAAL in a Nutshell », *International Journal on Software Tools for Technology Transfer*, vol. 1, n° 1-2, p. 134-152, 1997.
- [MC. 92] MC.MILLAN K. L., PROBST D. K., « A Technique of State Space Search Based on Unfolding », *Formal Methods in System Design*, p. 45-65, 1992.

- [PAR 06] PARK S., KWON G., « Avoidance of State Explosion Using Dependency Analysis in Model Checking Control Flow Model », *ICCSA* (5), p. 905-911, 2006.
- [PEL 98] PELED D., « Ten Years of Partial Order Reduction », *CAV '98 : Proceedings of the 10th International Conference on Computer Aided Verification*, Springer-Verlag, p. 17–28, 1998.
- [QUE 82] QUEILLE J.-P., SIFAKIS J., « Specification and verification of concurrent systems in CESAR », *Proceedings of the 5th Colloquium on International Symposium on Programming*, Londres, Royaume-Uni, Springer-Verlag, p. 337–351, 1982.
- [ROG 06] ROGER J.-C., Exploitation de contextes et d'observateurs pour la validation formelle de modèles, PhD thesis, ENSIETA, Univ. of Rennes I., décembre 2006.
- [VAL 91] VALMARI A., « Stubborn sets for reduced state space generation », *Proceedings of the 10th International Conference on Applications and Theory of Petri Nets*, London, UK, Springer-Verlag, p. 491–515, 1991.
- [WHI 06] WHITTLE J., « Specifying precise use cases with use case charts », *MoDELS'06, Satellite Events*, p. 290-301, 2006.

Chapitre 10

Déploiement et génération de code à partir du modèle

10.1. Introduction

L'objectif de ce chapitre est de décrire le processus de génération d'un binaire exécutable à partir du modèle de conception du pacemaker décrit dans le chapitre 8. Ce modèle de conception est un modèle à base de composants exécutables spécifiant la structure, le comportement interne et les interactions entre ces composants.

La structure est décrite par la décomposition du système en composants présentant des points d'interaction appelés ports. Les interactions sont modélisées par la mise en relation des composants par des connecteurs entre les ports de ces composants. La bonne formation du modèle d'interaction entre ces composants réside dans la compatibilité des ports mis en relation. Le comportement des composants est spécifié sous deux aspects : l'aspect algorithmique qui décrit les services offerts par le composant (ses opérations) et l'aspect contrôle qui décrit l'orchestration des appels à ces services ainsi que les différents modes de fonctionnement du composant. L'aspect contrôle de chaque composant est modélisé par une machine à états-transitions. L'aspect algorithmique est, quant à lui, spécifié par un langage d'actions.

A ce modèle de composants exécutables viennent s'ajouter d'une part un modèle de description de la plate-forme d'exécution et un modèle de description du déploiement des composants sur la plate-forme. Le modèle de description de la plate-forme permet

Chapitre rédigé par Chokri MRAIDHA, Ansgar RADERMACHER et Sébastien GÉRARD.

d'identifier le nombre de nœuds d'exécution offerts par la plate-forme. Cette information est utilisée pour la description du déploiement des composants sur ces nœuds. Ce déploiement permet d'identifier le nombre de binaires exécutables à générer tout en définissant le contenu de chacun de ces binaires.

Le modèle de composants exécutables est constitué de composants, de ports, de connecteurs, de machines à états-transitions, d'activités qu'il est nécessaire de traduire dans un langage de programmation de troisième génération (tout en conservant la sémantique d'exécution de ces concepts) afin de produire les binaires exécutables. Il est important de noter qu'aucun de ces concepts de modélisation ne possède de concept équivalent dans un langage de programmation. Par exemple aucun langage de programmation de troisième génération n'offre le concept de machine à états-transitions. En conséquence, la définition d'une projection des machines à états-transitions dans un langage de programmation nécessite la mise en œuvre de patrons appelés patrons de génération de code.

Afin de limiter la complexité des patrons de génération de code, il est important de choisir un langage de programmation cible offrant des concepts les plus proches possible du langage de modélisation, UML dans notre cas. De part l'historique et les concepts offerts par UML, les langages de programmation orienté objet s'avèrent être les langages cibles les plus adaptés à notre besoin. En effet ceux-ci offrent le concept de classe sur lequel se base la définition du concept de composant MARTE (voir chapitre 7). Aussi, nous utiliserons dans ce chapitre le langage C++ comme langage de programmation cible pour la génération de code.

La figure 10.1 décrit les principales étapes pour la génération d'un binaire exécutable à partir d'un modèle exécutable à composants, d'un modèle de description de plate-forme et d'un modèle de description du déploiement.

La bonne formation syntaxique et sémantique des modèles d'entrée est un préalable à tout traitement automatique de ceux-ci. Pour cette raison, la première étape du processus de génération d'exécutables, décrit par la figure 10.1, consiste à valider ces modèles. Un ensemble de règles de bonne formation sont ainsi vérifiées pour s'assurer que ces modèles sont cohérents et contiennent l'information nécessaire et suffisante à leur traitement automatique par la chaîne de génération. Dans le cas contraire les erreurs sont localisées et remontées au concepteur afin qu'il modifie ou complète les modèles d'entrée avant de refaire une validation.

La seconde étape du processus consiste à générer un modèle d'implantation à partir des modèles d'entrée valides. Ce modèle d'implantation est un modèle à composants dans lequel des mécanismes d'implantation sont explicités. Des patrons d'implantation concernant la communication et le comportement sont appliqués. A l'issue de cette étape, un modèle d'implantation global à tous les nœuds d'exécution est généré.

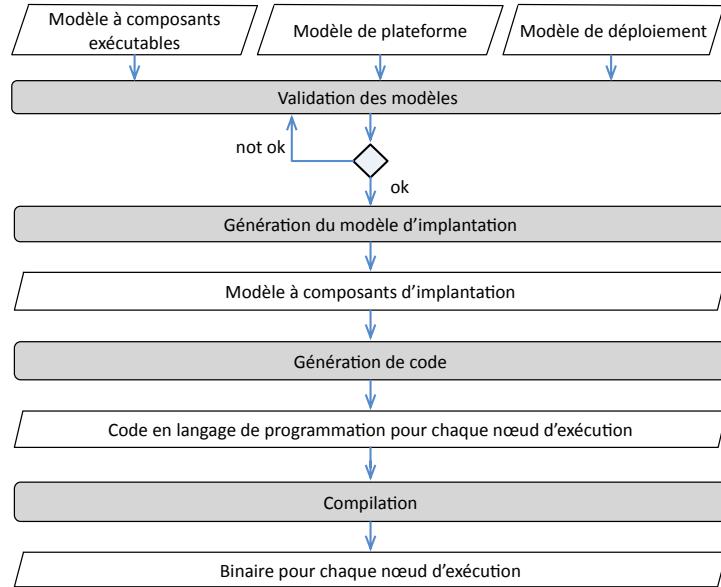


Figure 10.1 – Vue globale de l’approche

Ce modèle est fourni en entrée à l’étape de production de code en langage de programmation orienté objet pour chaque nœud d’exécution. Dans une dernière étape, ce code est compilé pour produire un binaire exécutable pour chacun des nœuds d’exécution de la plate-forme.

Ce chapitre est organisé comme suit. Une première section fournit sur la base du modèle du pacemaker une description des modèles d’entrée. La seconde section décrit l’étape de génération du modèle à composants d’implantation alors que la troisième section fournit les détails de la génération de code en décrivant les activités et les modèles intermédiaires pour cette étape. Un support outillé de l’approche est présenté dans une quatrième section. Quelques perspectives de travaux sont données en guise de conclusion.

10.2. Les modèles d’entrée

Cette section présente les modèles d’entrée du processus que sont le modèle à composants, le modèle de plate-forme et le modèle de déploiement.

10.2.1. Description du modèle exécutables à composants

Le modèle à composants MARTE utilisé correspond au modèle présenté en section 8.3. Ce modèle décrit les composants (avec leur comportement interne), les points d’interaction des composants (ports) caractérisés par les données que ceux-ci véhiculent ou les services qu’ils fournissent ou requièrent, ainsi que la topologie d’interconnexion des composants spécifiée par des connecteurs entre les ports.

Le modèle d’interaction, c’est-à-dire le type de communications, est spécifié par ce modèle à composants. Les types de communications sont :

- synchrone ou asynchrone pour les appels d’opération (ports client-serveur *clientServerPort*) ;
- *push* ou *pull* pour les communications de données (port de données *flowPort*).

Ces types d’interactions seront réalisés par des patrons d’implantation décrits en section 10.3.

Le modèle à composants spécifie également un modèle de concurrence de haut niveau en utilisant le sous-profil HLAM (*High Level Application Modeling*) de MARTE. Ce modèle de concurrence identifie les composants possédant des ressources d’exécution (RtUnit) et les composants partagés (PpUnit) ne possédant pas de ressources d’exécution mais des ressources gérant leur accès concurrent.

La figure 10.2 fournit un extrait du modèle à composants d’entrée du pacemaker. Ce système est constitué de deux composants principaux : le DeviceControlMonitor et le PGController. Ces deux composants communiquent au travers de ports ClientServer.

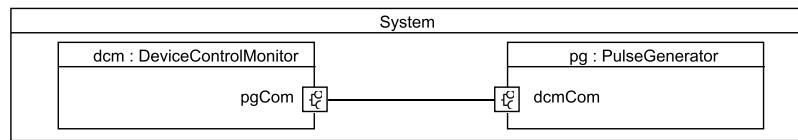


Figure 10.2 – Modèle à composants du système pacemaker

10.2.2. Description du modèle de plate-forme

De manière similaire à une architecture logicielle, une architecture matérielle est décrite par une composition d’éléments. Une classe, nommée « HWArchitecture » représente l’ensemble d’une plate-forme. Les attributs de cette classe représentent les noeuds. Chaque noeud est typé avec une classe contenant les propriétés de ce noeud. Chaque noeud peut avoir une structure interne, la structure hiérarchique pouvant ainsi être décomposée.

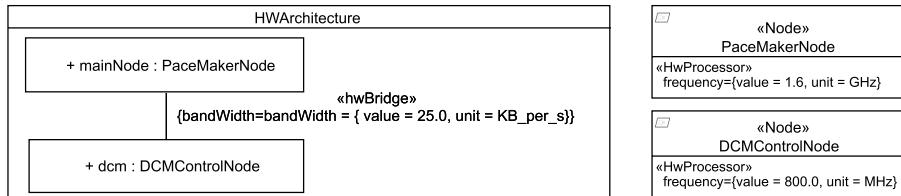


Figure 10.3 – Description de la plate-forme d'exécution du pacemaker

La figure 10.3 décrit la définition de la plate-forme pour le cas d'étude. Les composants de cette plate-forme sont modélisés et caractérisés en utilisant les concepts du sous-profil HRM (*Hardware Resource Modeling*) de MARTE. La plate-forme est constituée de deux nœuds d'exécution (PaceMakerNode et DCMControlNode) stéréotypés HwProcessor et d'un bus de communication stéréotypé HwBridge. Ces stéréotypes permettent d'associer des propriétés non fonctionnelles à ces composants matériels. Par exemple, le bus de communication fournit une bande passante de 25 kb/s.

10.2.3. *Description du modèle de déploiement*

Le déploiement d'une application consiste à définir les instances de composants, leur configuration ainsi que leur allocation sur un nœud d'exécution.

La structure composite UML utilisée pour la spécification du modèle de composants décrit plus haut, définit les rôles joués par chaque composant dans le système. Un rôle étant spécifié au niveau type, il se distingue d'une instance. Or les entités qui s'exécuteront, donc les entités à déployer, sont les instances de ces composants et non les composants eux-mêmes. En UML, une instance d'un composite est définie par une « InstanceSpecification », les instances des propriétés du composant sont instanciées par des « slots » auxquels des valeurs sont associées. Dans le cas où une propriété est typée par un composant, la valeur associée au « slot » est une instance de ce composant. Cette structure de décomposition hiérarchique nécessite une instanciation sur plusieurs niveaux souvent fastidieuse à construire à la main. Un support outillé pour la génération de cette arborescence d'instances facilite cette tâche.

Un système à base de composants nécessite souvent une configuration. Les attributs de configuration peuvent par exemple définir la fréquence d'un composant « Timer » pour décrire le déclenchement des événements liés à ce « Timer ». Les valeurs de ces attributs de configuration peuvent être définies à deux niveaux :

- au niveau de la déclaration de l'attribut en lui associant une valeur par défaut ;
- au niveau des instances où une valeur est spécifiée pour le slot correspondant.

Une fois que les composants sont instanciés et configurés, l’allocation des instances de composants peut se faire.

A gros grain, la phase d’allocation consiste à définir la relation entre les instances de composants logiciels et les ressources matérielles de la plate-forme. La figure 10.4 décrit le déploiement souhaité des composants DeviceControlMonitor et PGController sur la plate-forme matérielle. La relation « Allocate » fournie par MARTE est utilisée pour la déclaration de ce modèle d’allocation. La génération des instances et du déploiement des instances de ces composants sera réalisée lors de la phase de génération de code décrite dans la section 10.4.

L’allocation peut également se faire plus finement en introduisant des ressources logicielles de plate-forme dans un niveau intermédiaire entre les composants applicatifs et la plate-forme matérielle. Les ressources logicielles d’exécution sont les tâches modélisées en MARTE par le concept de « SwSchedulableResource ». Le modèle d’allocation consisterait alors à allouer les instances des composants applicatifs sur les ressources logicielles d’exécution, puis d’allouer ces ressources logicielles d’exécution sur les ressources matérielles d’exécution. La relation « Allocate » de MARTE est utilisée pour la capture de ces deux niveaux d’allocation. Une prise en compte plus fine de la plate-forme logicielle est également possible (gestion des ressources partagées, etc.). Le degré de spécificité à la plate-forme du modèle d’implantation décrit dans la section suivante dépend du niveau de détail du modèle de plate-forme donné en entrée.

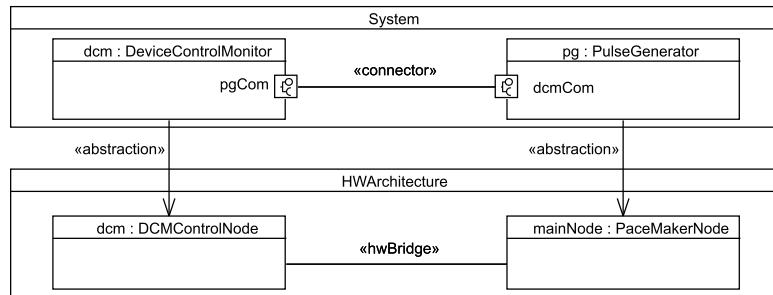


Figure 10.4 – Déploiement de composants logiciels sur la plate-forme

10.3. Génération du modèle d’implantation

Cette section décrit les deux principaux patrons de conception appliqués au modèle de composants applicatif d’entrée pour la génération d’un modèle à composants d’implantation. Ce dernier détaille la manière de réaliser les interactions entre les composants ainsi que les aspects techniques tels que la protection des composants passifs

(concept de PpUnit en MARTE). Les transformations proposées exploitent l’information déclarée dans le modèle d’entrée qui spécifie l’utilisation des mécanismes d’interaction (connecteur) et des services du container. L’objectif derrière ces deux mécanismes est la séparation des préoccupations : la fonctionnalité d’un composant doit être spécifiée indépendamment des contraintes techniques de l’environnement dans lequel il est embarqué. Une première partie de cette section introduit les principaux concepts mis en œuvre dans ces transformations.

10.3.1. Principaux concepts

Les principaux concepts nécessaires à la génération du modèle d’implantation sont : les composants, les connecteurs et les containers.

Un composant est une entité logicielle représentée par une classe UML. Un composant possède des points d’interaction appelés ports. Il doit explicitement déclarer quels services sont offerts et quels services sont requis par le composant à travers un port. Afin de maximiser la réutilisation, un composant ne doit pas connaître les composants offrant les services qu’il requiert. Un composant peut être avoir une structuration interne explicite en sous-composants aussi appelés composants d’assemblage.

Ces composants d’assemblage sont représentés par des *parts* ou propriétés UML. Ces propriétés sont typées par des composants (classes UML) et possèdent des ports qui peuvent être interconnectés avec les ports du composant englobant et/ou les ports des autres sous-composants. Un composant peut être un « type », c’est-à-dire une classe avec un ensemble de ports et pas de structure interne, ou une « implantation » qui réalise les services offerts par ses ports.

Un connecteur est un élément UML connectant au minimum deux *parts* ou deux ports d’une composition. En UML, un connecteur dénote une interaction entre deux éléments sans pour autant fournir d’informations sur la manière dont cette interaction est réalisée. Dans l’usage que nous en faisons, cette information est déclarée sur le connecteur (communication synchrone/asynchrone, etc.). Suite à l’application du patron connecteur, décrit dans la section suivante, un composant de réalisation de l’interaction sera généré dans le modèle à composants d’implantation.

Un container est une entité dont le rôle est d’encapsuler un composant (membrane externe) tout en prenant en charge les aspects non fonctionnels nécessaires à l’exécution des services du composant, laissant à ce dernier la seule charge de l’implantation du code métier.

10.3.2. Patron connecteur

Le patron de connecteur est basé sur l'idée du Garlan et Shaw [SHA 95] dans laquelle la réalisation d'une interaction est un élément de premier ordre du modèle. Une interaction peut, au même titre qu'un composant, avoir plusieurs réalisations possibles et être configurée. L'idée des connecteurs a été formalisée dans le cadre de UML dans [ROB 05a] et [ROB 05b] : un connecteur UML – un « fil » dans une classe composite – est remplacé par un composant d'interaction.

Le composant d'interaction est typiquement défini dans une librairie de modèles sous forme d'un *template*. Ce dernier est nécessaire, parce que le composant d'interaction doit être adapté à son contexte d'utilisation. Par exemple, dans le cas d'un appel d'une opération fournie par un composant, le composant d'interaction doit fournir cette opération. Principalement, nous considérons le cas où le paramètre du *template* est une interface ou un type de données (*data-type*). Dans le cas où c'est une interface, des ports fournis et requis du connecteur sont typés avec cette interface. Au niveau de la réalisation, l'implantation doit aussi être adaptée pour correspondre à l'interface. L'implantation d'un *template* est fournie sous forme d'un *template* textuel écrit dans un langage de transformation de modèle en texte (M2T).

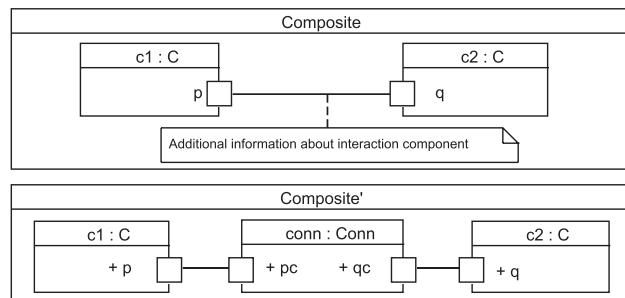


Figure 10.5 – Réification

Dans notre étude, la plupart des interactions sont des interactions basées sur des appels d'opérations, c'est-à-dire sans support particulier. Le traitement des signaux est différent, car il ne possède pas d'opérations qui peuvent être appelées. Un port qui permet l'émission d'un signal est donc associé à une interface dérivée qui permet son envoi :

```

interface D_Push_SensedPulse {
    push(in data : SensedPulse);
}

```

Le *template* est défini au niveau paquetage pour permettre la spécification d'un ensemble d'éléments qui dépend du paramètre formel. La figure 10.6 montre la spécification de deux types de connecteurs pour le flot de données (*data-flow*), un pour le modèle *push-push* et un pour le modèle *push-pull*. Dans le premier cas, le producteur joue un rôle actif (il produit une donnée à un certain instant) et le consommateur joue un rôle passif (il est appelé par l'environnement quand une donnée arrive). Dans le deuxième cas, le consommateur joue un rôle actif (il lit des données à un instant précis, c'est-à-dire *pulls for data*). Le paquetage définit deux implantations pour le modèle *push-pull*, un basé sur un file et un autre qui mémorise la dernière valeur reçue (une file de taille 1). L'héritage entre type et implantation s'exprime plus facilement avec le *template* au niveau paquetage : il n'est pas nécessaire de mélanger *binding* et héritage.

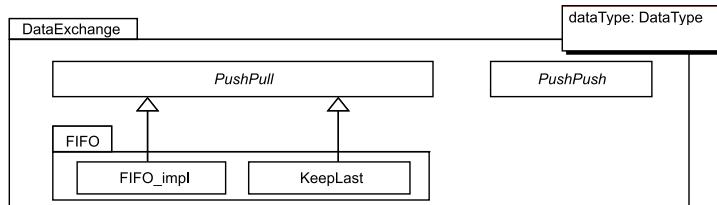


Figure 10.6 – Patron de paquet pour connecteurs FIFO et *KeepLast*

L'utilisation de paramètre formel est décrit dans la figure 10.7 : les ports du type de connecteurs sont typés avec le paramètre formel, et un attribut à l'intérieur de la file FIFO est typé avec ce paramètre. Le paramètre *template* apparaît aussi dans les opérations fournies par la FIFO (cela n'est pas visible dans le diagramme de structure composite), une fois dans la déclaration d'opération et une fois dans le comportement. Ce comportement peut être décrit avec des machines à états, des activités UML ou avec un langage de programmation classique (sous forme de comportement opaque, *OpaqueBehavior*). Pour la réalisation de FIFO, la dernière option a été utilisée. Un langage de transformation de modèle vers du texte, comme Acceleo peut être utilisé pour instancier le code (un bloc de texte) avec le paramètre actuel du *template*. La mise en œuvre de cette option sera décrite dans le cadre des machines à états dans la prochaine section.

10.3.3. Patron container

Un patron connu pour modifier la manière avec laquelle un objet agit avec son environnement est l'ajout d'un *container*. Ce patron a également été identifié par des intergiciels comme CCM [OMG 08] et Fractal [BRU 04]. Le container encapsule l'objet (composant) et peut fournir des services et observer ou manipuler les interactions du composant. Bien que le modèle à composants applicatif contienne l'information sur

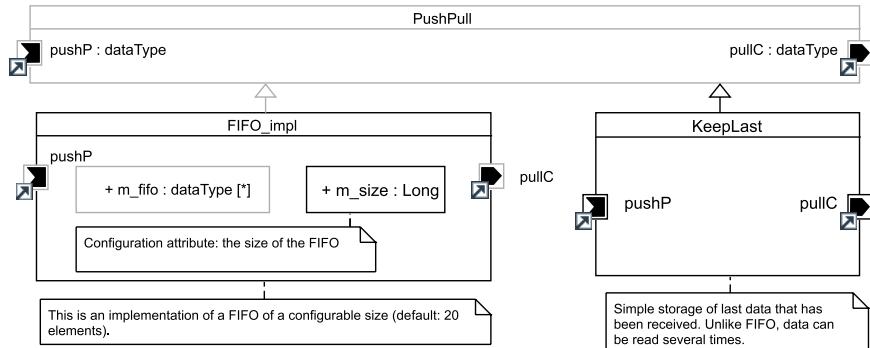


Figure 10.7 – Réalisation des connecteurs FIFO et KeepLast

le contenu à ajouter au container, il est préférable (pour plus de flexibilité) de ne pas ajouter directement au modèle applicatif les éléments du container. Ainsi, l'utilisateur final applique une règle de transformation, sans pour autant connaître en détail les éléments qui seront ajoutés au container.

La figure 10.8 montre ce principe : un composant « C » est enrichi avec les règles à appliquer. Cette information est évaluée par une transformation du modèle qui crée le container et ajoute les éléments associés avec les règles au container. Le composant devient un exécuteur, c'est-à-dire le code métier derrière un composant. Il est possible de distinguer deux types d'éléments dans le container : soit des intercepteurs, soit des extensions. L'intercepteur se place sur la connexion de délégation entre un port du container et un port du composant métier contenant le code à exécuter. L'extension est un élément en plus qui peut être connecté avec des ports externes du container (conformément à sa spécification dans le règle du container).

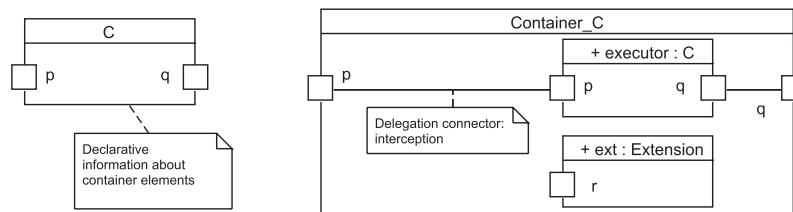


Figure 10.8 – Expansion d'un container

Les bibliothèques de container offrent par exemple la production de traces ou la réalisation de patron d'exclusion mutuelle (comme les sémaphores permettant la gestion d'accès concurrents à une ressource). Dans le contexte du pacemaker, un service de

container est particulièrement important : le support des machines à états. Une machine à états constitue une combinaison des trois éléments du container, la machine elle-même, un *pool* d'événements et un intercepteur qui fournit des événements associés aux appels d'opérations (*CallEvent*). La règle qui produit la machine à états dans le container est décrite dans la figure 10.9. Les trois éléments, intercepteur, *pool*, machine à états ainsi que leur connexions sont modélisés. De plus, un stéréotype sur l'intercepteur définit l'ensemble les ports interceptés : tous, un ensemble spécifié, tous les ports d'entrée ou tous les ports de sortie. Dans le cas où l'intercepteur couvre la machine à états, tous les ports sont interceptés et les intercepteurs produisent des événements qui correspondent aux appels transmis à l'exécuteur (« CallEvents » selon la norme UML).

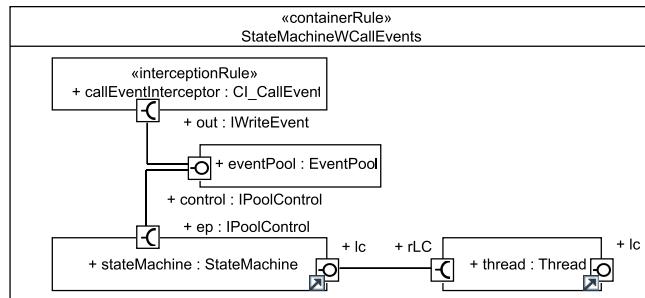


Figure 10.9 – Règle de container pour une machine à états

De manière similaire au connecteur FIFO, les machines à états et l'intercepteur sont définis dans les *templates* au niveau paquetage (figure 10.10). Le modèle de la machine à états est défini dans le composant (la classe), dans cet exemple d'étude la classe « Cardiologist ». L'instanciation de l'implantation qui peut exécuter une machine à états dépend donc d'un paramètre formel de type classe. L'intercepteur des messages est typé avec une interface car il est placé entre les ports typés avec une interface spécifique.

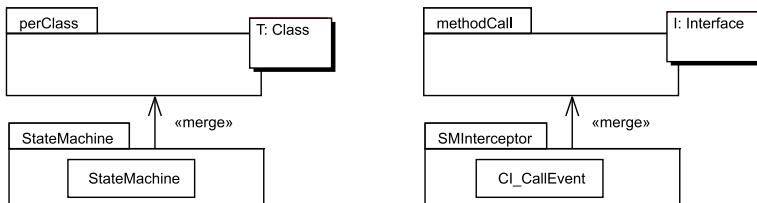


Figure 10.10 – Template d'une machine à états

```
// checkPreCond body - Generated by eC3M
PR ("IN [clazz.name]/::checkPreCond(): currentState : ",
     << [clazz.name/]_CsStr(m_currentState) << showI);
[for (sm : StateMachine | ownedBehavior->select(oclIsKindOf(StateMachine)))]
int newState;

switch(m_currentState)
{
    [for (state : State | sm.region.subvertex->select(oclIsKindOf(State)))]
    case [clazz.name/]_[state.name/]:
        ...
    [/for] ...
}
```

Figure 10.11 – Template pour la réalisation d’opération « accept »

```
PR ("IN Cardiologist::checkPreCond(): currentState : ",
     << Cardiologist_CsStr(m_currentState) << showI);
int newState;

switch(m_currentState)
{
    case Cardiologist_StartTherapy:
        ...
}
```

Figure 10.12 – Résultat de l’instanciation du template avec la classe « Cardiologis »

La figure 10.11 décrit le patron d’une opération qui (parmi d’autres) gère l’exécution d’une machine à états. Le code, spécifié ici en langage C++, est modélisé par un comportement opaque en UML. Les accès aux éléments du modèle et aux éléments de contrôle sont embarqués entre [et /], c’est-à-dire [name/] permet l’accès au nom d’un élément UML. Cet extrait de code montre la puissance d’adaptation du comportement des composants embarqués dans le container pour créer des nouvelles fonctionnalités bien séparées du code métier.

La figure 10.12 illustre le résultat d’instanciation avec la classe « Cardiologist » du cas d’étude. Cette classe contient une machine à états avec l’état « StartingTherapy », qui apparaît dans le « switch ». Dans chaque état, les événements définis comme déclencheurs (*triggers*) pour les transitions sont comparés avec l’évènement reçu par le *pool* (qui est alimenté par les intercepteurs ou un *timer*), et un nouvel état est assigné à l’état courant.

10.3.4. *Implantation des composants*

Une implantation de composants peut être fournie par des opérations dont le corps peut être modélisé avec une activité décrite par le langage d'action ALF ou avec un comportement opaque décrit dans un langage de programmation comme C++. Dans le cas d'un langage d'action, des commandes spécifiques pour interagir avec un port existent.

Pour les langages de programmation de troisième génération, comme C++, la notion de port est inexistante. Dans le cas d'une émission, une implantation peut alors appeler une opération associée à un port, ou dans le cas d'une réception (en mode *push*) une de ses opérations peut être appelée. Un port possède donc une association entre son type et une interface fournie et requise dérivée. On parle d'un *mapping* vers une interface requise et fournie. Par exemple : un *flow port* MARTE qui produit des données est « mappé » sur une opération « *push* » avec la donnée à transférer (type de port) en paramètre.

Un *flow port* qui consomme des données a deux réalisations différentes. Dans la première, la consommation est déclenchée par l'environnement, qui appelle une opération du consommateur (exécutée par une tâche de l'intergiciel ou appelant). Dans la seconde, le composant actif vérifie à un instant donné, si une donnée est arrivée à son port et l'exécute. Dans ce cas, le producteur et le consommateur possèdent des ports avec une interface requise. Pour le coupler, nous avons besoin d'une FIFO. Dans notre cas d'étude, la deuxième variante est majoritairement utilisée.

10.3.5. *Composants d'implantation résultants*

Le résultat d'exécution des deux transformations est un modèle d'implantation qui est encore un modèle à composants. Ce modèle explicite les solutions choisies pour les connecteurs et contient des containers pour tous les composants. Ce modèle est un modèle à composants dépendant de la plate-forme et décrivant la réalisation de ce qui a été déclaré dans le modèle à composants MARTE d'entrée. Il est le résultat de l'application des patrons de réalisation de composants, les patrons container et connecteurs.

La figure 10.13 illustre le modèle d'implantation résultant sur la classe « Cardiologist » du cas d'étude. Le container généré contient quatre intercepteurs, correspondant aux quatre ports de la classe. Chaque intercepteur a un type différent, qui correspond aux différentes instantiations du *template* au niveau paquetage. De plus, le container possède un *pool* d'événements et la machine à états instanciée.

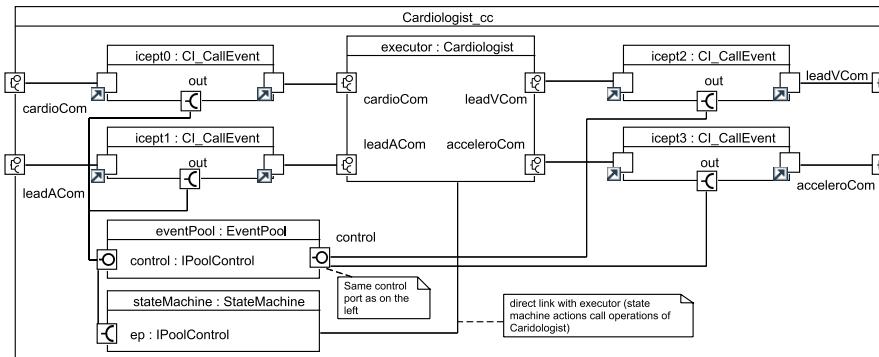


Figure 10.13 – Container généré pour la classe « Cardiologist »

10.4. Génération de code

Le résultat de la phase précédente est un modèle à composants de l’application, enrichi avec des connecteurs réifiés et des containers expansés. La génération de code à partir de ce modèle nécessite au préalable la mise en œuvre des deux actions suivantes :

- la réalisation du déploiement des composants. Cette réalisation consiste à découper le modèle global en sous-modèles pour chaque nœud d’exécution. Les dépendances des sous-modèles entre eux doivent être générées. Le résultat de cette activité est, conformément au déploiement déclaré en entrée, un modèle d’implantation pour chaque nœud d’exécution ;

- la transformation des notions de ports et de connecteurs qui n’ont pas d’équivalent direct en langage de programmation orienté objet (comme C++).

Les sous-sections suivantes détaillent ces deux activités sur l’exemple de l’étude de cas du pacemaker.

10.4.1. Déploiement des composants

Dans le cadre de cette activité, un modèle par nœud doit être produit. Pour ce faire les deux tâches suivantes sont nécessaires : une séparation des instances et une copie des éléments nécessaires à la réalisation de ces instances.

La première tâche peut sembler triviale dans le cas où chaque instance de composant est associé explicitement à un seul nœud. Cependant, des cas plus complexes peuvent se présenter. Ainsi, dans le cas de composants composites dont les sous-composants sont déployés sur des nœuds différents, le composant père se trouve implicitement alloué sur ces différents nœuds. Cela induit des contraintes sur les propriétés de ces

composants. Pour éviter les problèmes de gestion de cohérence des valeurs de ces propriétés, seules des propriétés de configuration en lecture seule sont autorisées. Ces contraintes de bonne formation sont vérifiées lors de la phase de validation des modèles d'entrée.

La seconde tâche consiste à compléter chacun des modèles par nœud avec les éléments dont ils dépendent pour leur exécution. Ces éléments peuvent être des types/implantations des attributs, classes héritées, etc. Le résultat de cette tâche est pour chaque nœud un modèle autonome dans le sens où il contient tous les éléments nécessaires à son exécution.

La figure 10.14 représente les modèles pour les nœuds d'exécution présentés dans la figure 10.4. Deux modèles d'application sont générés : un modèle pour le nœud sur lequel est déployé le composant PGController (et tous ses sous-composants), et un modèle pour le nœud sur lequel est déployé DeviceControlMonitor. Chacun de ces modèles contient une partie du composant global System et les types (DomainTypes) utilisés par ses composants.

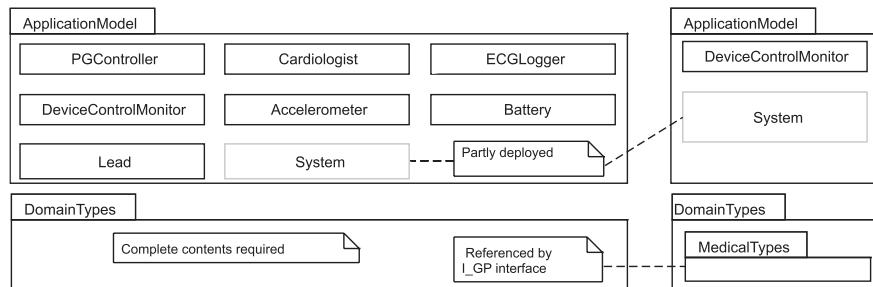


Figure 10.14 – Réalisation du déploiement

Ces modèles à composants exécutables peuvent maintenant être traduits en modèles orientés-objet pour une génération de code en langage de programmation orienté objet.

10.4.2. Transformation en modèle orienté objet

Les notions de port et de connecteurs simples (c'est-à-dire réifiés) ne possèdent pas de concepts équivalents en langage de programmation orienté objet. Il est nécessaire de transformer ces concepts en utilisant des patrons mettant en relation des concepts du paradigme orienté objet afin de produire des mécanismes équivalents aux ports et aux connecteurs dans ces langages. Les principaux concepts des langages de programmation orienté objet sont les classes, les interfaces, les attributs et les opérations.

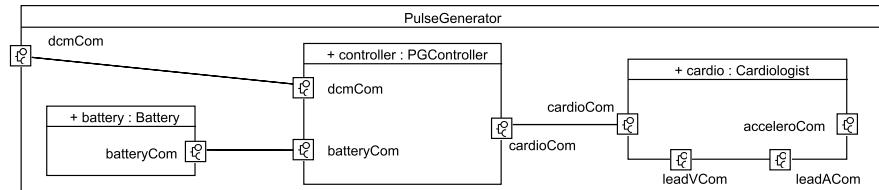


Figure 10.15 – Extrait de l'architecture à composants du générateur de pulsation

Nous distinguons les ports à interface fournie et les ports à interface requise (le cas, où un port offre et requiert des interfaces est également possible). Un port fournissant une interface est un point d'accès à un service. Pour la réalisation d'un composant fournissant ce service, il est important d'être capable d'obtenir une référence à ce port. Si un composant possède un port « p » qui fournit une interface « I », la réalisation d'un composant doit avoir une opération d'accès au port « get_p » qui retourne la référence pour ce port. L'implantation de cette opération est calculable de manière automatique : s'il existe un connecteur de délégation vers une *part* à l'intérieur du composant, cette référence est retournée ; sinon, la référence du composant lui-même est retornée.

Un port avec une interface requise est un point d'interaction qui nécessite une référence vers un autre composant fournissant l'interface (plus précisément le port du composant fournissant l'interface). Le composant doit donc enregistrer cette référence et fournir une opération pour initialiser la référence lors de l'instanciation. Chaque port « q » avec une interface requise est transformé en un attribut « q » (de type agrégation « shared ») qui stocke la référence vers un port fournissant l'interface et une opération « connect_q » servant à la configuration du composant. La figure 10.16 montre la classe résultant de cette transformation pour le composant « PGController », présenté dans la figure 10.15.

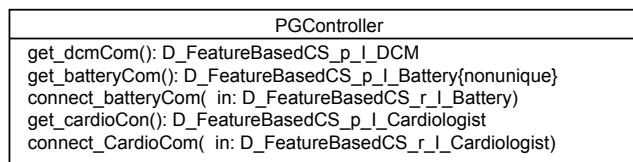


Figure 10.16 – Transformation en paradigme orienté objets

Le code ci-dessous montre le code généré pour les opérations `get_dcmCom` et `createConnections()` pour l'architecture présentée dans la figure 10.15. Le premier est associé au connecteur de délégation entre le `PulseGenerator` et le `controller` dedans. Il est à noter que certaines connexions sont bidirectionnelles, par exemple la connexion entre `battery` et `controller`. Les ports associés sont typés avec des ports client/server de MARTE

qui fournissent et requièrent des opérations. Il y a donc deux paires d’interfaces fournies requises et deux connexions.

```
void PulseGenerator::get_dcmCom() {
    // realization of delegation connector
    return controller.get_dcmCom();
}

void PulseGenerator::createConnections() {
    // realization of connector <controller-cardio>
    controller.connect_cardioCom (cardio.get_cardioCom());

    // realization of connector <battery-controller>
    controller.connect_batteryCom (battery.get_batteryCom());
    battery.connect_batteryCom (controller.get_batteryCom());
    ...
}
```

10.4.3. *Code en langage de programmation*

Comme les transformations de modèles à composants vers des modèles orientés objet sont déjà effectuées, un générateur de code classique prenant en entrée un modèle UML orienté objet est donc suffisant pour la génération de code en langage de programmation orienté objet (C++ dans notre cas). Pour chacune des classes ou interfaces, une classe C++ est générée. Les paquetages de UML sont traduits par des déclaration de « namespace » en C++. L’organisation des fichiers suit le même schéma que celui qui serait appliqué en Java. Un *namespace* correspond à un répertoire et la structure de fichiers reflète donc la hiérarchie des paquetages du modèle UML.

Les dépendances vers des paquetages externes sont traduits par des liens « include » vers des bibliothèques. Le code C++ généré peut être compilé dans un environnement de compilation C++.

10.5. Support outillé

eC3M (*embedded Component Container Connector Middleware*) [RAD 09] est un environnement supportant le processus décrit dans les sections précédentes. eC3M offre les règles de validation ainsi que les transformations de modèles décrites pour la génération d’un code exécutable à partir d’un modèle à composants. eC3M est intégré à l’outil de modélisation UML Papyrus. La figure 10.17 donne une vue globale de l’environnement eC3M.

Les modèles d’entrée sont des modèles UML qui appliquent les profils UML MARTE et FCM [JAN 11]. FCM est un profil qui permet de spécifier l’information de réalisation nécessaire sur les ports, les connecteurs et les composants. Un port en FCM est

caractérisé par un type et une sorte (*kind*). La sorte sert à sélectionner les règles de traduction vers des interfaces fournies et requises, comme cela a été décrit en section 10.3.4. Pour les connecteurs en UML, le type ou l'implantation sont décrits avec FCM. L'application du stéréotype FCM enregistre une référence vers une bibliothèque de modèles contenant des connecteurs. Pour les composants, FCM permet de faire la distinction entre type et implantation. De plus, FCM permet l'application d'un ensemble de règles de containers pour les composants. De nouveaux types de connecteurs et de containers peuvent être définis dans des bibliothèques. Ces bibliothèques constituent ainsi un moyen d'extension de l'environnement eC3M.

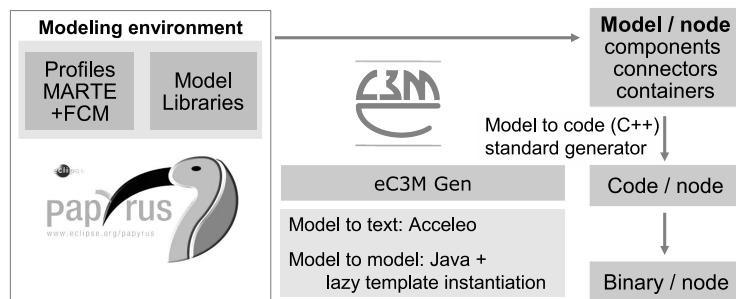


Figure 10.17 – Chaîne outillée eC3M

A partir d'un modèle UML/MARTE/FCM, l'environnement eC3M réalise un ensemble de transformations de modèles, notamment la réification des connecteurs, l'extension des containers, une distribution du modèle sur des noeuds et la transformation du modèle à composants vers un modèle orienté objet – comme précédemment décrit. La génération de code est basée sur un générateur modèle vers du code standard, ici du C++. La compilation utilise également des compilateurs standards, dans notre cas le compilateur gnu g++. L'intégration dans l'environnement Eclipse est réalisée par l'utilisation de CDT, le C++ development tools pour Eclipse. eC3M génère un projet CDT par noeud et configure les chemins (-I et -L) pour trouver les bibliothèques externes requises. Cette chaîne permet donc la génération d'un binaire par noeud. eC3M supporte actuellement un déploiement statique, c'est-à-dire un déploiement sans allocation dynamique. Il est donc suffisant de copier chaque binaire sur son noeud pour exécuter l'application.

10.6. Conclusion

Dans ce chapitre nous avons présenté un processus de génération de binaires exécutables à partir d'un modèle MARTE à composants distribués. Les principales étapes de cette chaîne de génération sont : la validation des modèles d'entrée, la génération d'un modèle d'implantation et la génération de code.

10.7. Bibliographie

- [BRU 04] BRUNETON E., COUPAYE T., STEFANI J., The Fractal Component Model, 2004, fractal.objectweb.org/specification/.
- [JAN 11] JAN M., JOUVRAY C., KORDON F., KUNG A., LALANDE J., LOIRET F., NAVAS J., ANDJ. PULOU L. P., RADERMACHER A., SEINTURIER L., « Flex-eWare : a flexible model driven solution for designing and implementing embedded distributed systems », *Software : Practice and Experience*, vol. 42, n° 6, 2011.
- [OMG 08] OMG, CORBA Component Model Specification, Version 4 (part of the CORBA 3.1 Specification), 2008, OMG Document formal/2008-01-08.
- [RAD 09] RADERMACHER A., CUCCURU A., GERARD S., TERRIER F., « Generating Execution Infrastructures for Component-oriented Specifications With a Model Driven Toolchain – A case study for MARTE’s GCM and real-time annotation », *Eighth International Conference on Generative Programming and Component Engineering (GPCE’09)*, ACM press, p. 127–136, 2009.
- [ROB 05a] ROBERT S., RADERMACHER A., SEIGNOLE V., GÉRARD S., WATINE V., TERRIER F., « Enhancing Interaction Support in the CORBA Component Model », RETTBERG A., ZANELLA M. C., RAMMIG F. J., Eds., *From Specification to Embedded Systems Application*, IFIP TC10 Working Conference : International Embedded Systems Symposium (IESS), Springer, 2005.
- [ROB 05b] ROBERT S., RADERMACHER A., SEIGNOLE V., GÉRARD S., WATINE V., TERRIER F., « The CORBA Connector Model », *Proceedings of the 5th International Workshop on Software Engineering and Middleware*, ACM Digital Library, 2005.
- [SHA 95] SHAW M., GARLAN D., *Software Architecture : Perspectives on an Emerging Discipline*, Prentice Hall, 1995.

QUATRIÈME PARTIE

AADL

Chapitre 11

Présentation des concepts de AADL

11.1. Introduction

Ce chapitre présente le langage de description d'architecture AADL (*Architecture Analysis and Design Language*) dans sa version 2. AADL est un standard défini par la *Society of Automotive Engineers* (SAE) [SAE 09]. Il fait partie de la famille des langages d'architecture, permettant de structurer une ou plusieurs implantations d'un système en vue de son analyse par différentes techniques (simulation, preuve formelle, *model-checking*), voire de sa réalisation.

Nous introduisons dans la section 11.2 les concepts généraux des ADL, puis en section 11.3 une présentation haut niveau de AADLv2 et les outils associés. Nous détaillons le langage dans la section 11.4 et ses annexes en section 11.5. Enfin, nous terminons par un bref aperçu des analyses supportées et outillées par AADL en section 11.6.

11.2. Concepts généraux des ADL

De nombreux langages de description d'architecture (ou ADL) ont été proposés, chacun fournissant des niveaux d'abstraction différents, en lien avec des besoins de documentation, d'analyse ou de génération de code. Cela fait que la définition précise d'un ADL, en opposition à un langage de programmation classique, est longtemps restée floue. Les travaux de Medvidovic et Taylor [MED 00] ont permis une première classification des éléments d'un ADL, que l'on peut résumer comme suit : « Un ADL

Chapitre rédigé par Jérôme HUGUES et Xavier RENAULT.

permet de décrire l'architecture d'un système sous la forme d'un ensemble de composants liés par des connecteurs décrivant les mécanismes d'interaction, définissant ainsi un configuration du système.

Un composant est une unité de composition dont l'interface et le contexte d'exécution sont complètement spécifiés, par exemple une tâche Ada, un composant Java EJB, etc. Les connecteurs définissent la sémantique d'interaction entre les interfaces de deux composants, par exemple un appel de procédure, un envoi de message, etc. La configuration du système est un assemblage de composants et de connecteurs syntaxiquement et sémantiquement cohérent.

Ces définitions étant posées, nous pouvons présenter les concepts de base de AADL.

11.3. AADLv2, un ADL pour la conception et l'analyse

Le langage de description d'architecture AADL, basé sur la description de composants, est standardisé par la SAE (*Society of Automotive Engineers*) [SAE 09]. AADL permet de modéliser à la fois les aspects logiciels et matériels d'un système temps réel en supportant une démarche de modélisation par raffinement (héritage simple), avec une sémantique clairement définie conforme aux systèmes avioniques et spatiaux, et un mécanisme avancé de gestion de propriétés non fonctionnelles.

11.3.1. Historique d'AADL

AADL a une longue histoire, héritée de plusieurs projets successifs ayant cherché à clarifier le périmètre d'un ADL. Ce long processus de maturation garantit non seulement un pouvoir d'expression élevé, mais aussi que le modèle ainsi construit peut être analysé de manière automatique par un outil. AADL découle de deux ADL « historiques » dont il a emprunté plusieurs concepts : MetaH et ACME.

MetaH [FEI 00] est un ADL initié par Honeywell Technology sous l'impulsion de la DARPA. Son auteur principal, Steve Vestal, s'est inspiré de la syntaxe du langage Ada pour sa représentation textuelle. MetaH disposait des concepts explicites pour modéliser tâches, sous-programmes, processeurs, etc. Ainsi que des propriétés non fonctionnelles leur étant associées. MetaH a été déployé dans plusieurs projets militaires américains, notamment des systèmes avioniques. MetaH disposait à la fois d'un environnement de modélisation, d'outils d'analyse et de génération de code vers des automates hybrides, et couvre ainsi les activités du cycle de vie du logiciel.

ACME [SCH 04] est un ADL développé au *Software Engineering Institute* (SEI) de l'Université Carnegie Mellon, disposant de capacités d'extension et de raffinement des composants, ainsi qu'une large palette d'outils d'analyse. En revanche, il ne dispose pas de mécanismes spécifiques aux systèmes temps réel, et reste donc générique.

La manipulation explicite de concepts proches de l'ingénieur fourni par MetaH, et les capacités de modélisation par extension et raffinement de ACME sont les deux blocs fondateurs d'AADL. Ils permettent de modéliser à la fois des systèmes à large échelle, tirant partie des capacités d'extension, proches des modélisations objet, mais aussi des systèmes précis, pour lesquels tous les éléments pertinents pour l'analyse ou la génération de code sont présents dans le modèle.

Le travail de standardisation d'AADL a démarré auprès du SAE en 1999, afin de rendre disponible un standard ouvert, permettant la construction de suites d'outils intégrées. Le but du langage AADL est de permettre simultanément la conception d'architecture, leur analyse (vérification et validation) et la génération de systèmes, dans un contexte temps réel. L'objectif est de disposer d'une syntaxe textuelle, d'une représentation graphique et d'un format d'échange basé sur XML, découlant d'un métamodèle.

La première version du standard a été publiée en 2004, suite à un effort conjoint du SEI, DoD (*Department of Defense*), Lockheed Martin, Airbus et l'ESA (Agence spatiale européenne). Le comité AADL s'est ensuite étendu, rejoint par la communauté académique qui a défini des passerelles entre outils d'analyse formels et AADL.

Une seconde version du standard a été publiée en 2009, étendant d'une part le pouvoir d'expression du langage, permettant davantage de modularité (utilisation renforcée des paquetages, modélisation paramétrique, composants abstrait, etc.). Par ailleurs, de nouvelles annexes ont été définies. Elles fournissent des guides pour la modélisation avancée des données, du comportement de composants ou encore des erreurs, de systèmes avioniques basés sur le standard ARINC653[ari07]. C'est cette seconde version que nous présentons ici.

Il est important de souligner que le développement d'AADL s'est fait en liaison avec plusieurs projets de recherche et développement combinant utilisateurs académiques et industriels, parmi lesquels IST-ASSERT, TopCased, SPICES, l'initiative SAVI de l'*Aerospace Vehicle Systems Institute (AVSI)*¹. Les évolutions du langage ont tenu compte des différents besoins exprimés par ces projets.

11.3.2. Une introduction rapide à AADL

Dans cette section, nous présentons les éléments fondamentaux d'un modèle AADL, que nous présenterons plus en détail par la suite.

1. Voir https://wiki.sei.cmu.edu/aadl/index.php/The_Story_of_AADL pour plus de détails sur ces projets.

La figure 11.1 fournit une vision haut niveau des différents blocs formant un modèle AADL. Ceux-ci sont formés d'ensemble de propriétés (*property sets*) et de paquetages. Les ensembles de propriétés définissent les attributs pouvant être attachés à des éléments du modèle : nom, type, unité. Les paquetages rassemblent les éléments du modèle : types des composants, et implantations de composants.

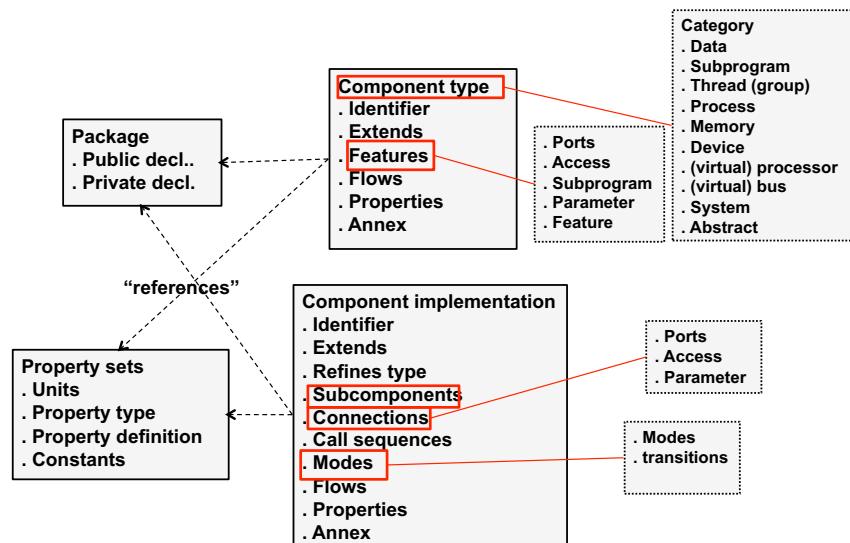


Figure 11.1 – Vision simplifiée des éléments d'AADLv2

– Les types de composants définissent les signatures des composants formant les briques de base de l'architecture : son identifiant, le composant qu'il étend par un mécanisme d'héritage simple, son interface et ses propriétés. Les types correspondent à une catégorie prédéfinie (tâches, processeurs, etc.) ; les interfaces à des mécanismes d'interaction connus : échange de messages, d'événements, de données, etc.

– Les implantations de composants définissent les réalisations concrètes des types ; ses sous-composants, et les connexions entre ces sous-composants et l'interface du composants. Une implantation dispose de son propre jeu de propriétés.

Au sein d'un composant, les modes permettent de modéliser les différentes configurations d'un système, les décisions de reconfiguration en cas d'événement prévu (par exemple un cas d'erreur ou un ordre de démarrage). Chaque configuration peut spécifier des valeurs différentes pour les propriétés, mais aussi activer/désactiver certains composants. Ainsi, les modes définissent l'ensemble statique de configurations du système.

Se faisant, AADLv2 est donc un ADL au sens de la taxonomie de [MED 00].

Les différents éléments d'un modèle doivent être regroupés en paquetage, définissant autant d'espaces de noms différents. Les dépendances entre paquetage doivent, à la manière d'un langage de programmation, être rendues explicites lors de la modélisation.

AADL définit un jeu de règles syntaxiques et sémantiques permettant de garantir la cohérence des modèles et leur bonne composition. Ces règles garantissent que le modèle est formé d'un assemblage valide de composants, mais ne libère pas des activités de vérification et validation supportées par des outils ou une relecture du modèle.

11.3.3. Outils

Un langage de modélisation sans outil a un intérêt plus que limité. Afin de tester le langage de manière détaillée, AADL a été implanté dans différents outils, dont plusieurs sont des logiciels libres :

- OSATE2 : le SEI développe une implantation de référence d'un outil AADL, OSATE [CMU 09], sous forme d'une extension Eclipse sous licence libre. OSATE dispose d'un frontal supportant le langage AADLv2 ainsi qu'un éditeur de texte avancé. En tirant parti de la technologie Eclipse, OSATE a permis le développement de nombreuses passerelles entre AADL et des outils d'analyse : outil d'optimisation de modèle, d'analyse de sécurité ou de calcul de latence sur les flots ;

- AADLInspector : AADLInspector est un logiciel développé par Ellidiss Technologies. Il fournit un éditeur texte léger pour l'édition de modèles AADLv2 textuels, ainsi que des connexions vers différents outils d'analyse et de simulation. Il repose sur Cheddar [SIN 07] pour l'analyse d'ordonnancement et un outil permettant la simulation de modèles AADLv2 complétés par une description comportementale ;

- Ocarina : Ocarina [LAS 09] est une suite d'outils conçus en Ada par l'équipe S3 de TELECOM ParisTech, auquel contribuent maintenant l'Institut supérieur de l'aéronautique et de l'espace (ISAE) et l'ESA. Ocarina est conçu comme un compilateur traditionnel divisé en deux parties : un frontal et une famille de dorsaux pour la génération de code et de modèles d'analyse. Ainsi, Ocarina fournit un générateur de code pour les intergiciels PolyORB-HI C et Ada, les réseaux de Petri colorés ou temporisés, des annotations pour l'analyseur de pire temps d'exécution (WCET) *Bound-T*, des modèles de tâches pour l'analyse d'ordonnancement avec Cheddar, et un langage de contraintes comme langage d'annexes, REAL, permettant de contraindre les patrons de modélisation utilisés.

le singulier me paraît mieux, on génère du code, ...

11.4. Taxonomie des entités AADL

Dans cette section, nous présentons les différents éléments du langage AADL, dans ces notations graphiques et textuelles.

Retenant les éléments canoniques d'un langage de description d'architecture, AADL définit la notion de composants et de connexions. Les connexions sont une vue simplifiée des connecteurs pour lesquels seules certains politiques sont supportés. Composants et connecteurs sont liés dans les composants d'implantation pour former une configuration. Composants, connexions et configurations peuvent être annotées de propriétés affinant leur description.

Dans la terminologie AADL, une propriété est un élément caractéristique par sa valeur d'un composant AADL : priorité d'un tâche, politique d'ordonnancement, taille mémoire, politique de déclenchement de tâche, etc. Il diffère donc de la notion de propriété d'un système, qui est obtenu par analyse d'un sous-ensemble du modèle, tels que la propriété d'être ordonnable, sûr, etc.

11.4.1. *Eléments de langage : composants*

AADL différencie deux familles de composants : les composants logiciels et les composants matériels.

Les premiers permettent de décrire les éléments applicatifs d'une architecture. Les seconds décrivent les éléments sur lesquels les éléments applicatifs seront déployés. Ces composants sont organisés de façon hiérarchique dans des composants hybrides dits « systèmes », qui peuvent eux-mêmes en contenir d'autres, ou « abstraits » lorsque la catégorie du composant n'est pas encore connue.

Composants logiciels. On distingue plusieurs types de composants logiciels :

- les tâches légères (*threads*) : les tâches modélisent une tâche concurrente ou un objet actif. Il s'agit d'un élément ordonnable pouvant s'exécuter de façon concurrente avec d'autres tâches rattachés au même ordonnanceur. Il s'exécute toujours dans l'espace mémoire d'un processus et se déclenche soit de façon périodique sur un événement temporel, soit sur l'arrivée de données ou d'événements sur leurs interfaces (*ports*). Ceux-ci sont gelés une fois l'exécution déclenchée : aucun événement ou donnée ne peut être pris en compte jusqu'au prochain déclenchement ;
- les groupes de tâches : ils permettent de grouper logiquement les tâches d'un système (*pool* de tâches par exemple). Un groupe de tâches est contenu dans un composant système et peut contenir, outre les tâches, des composants de données (variables partagées par les tâches), ou encore des composants représentant des sous-programmes (appelés par les tâches du groupe, lors de leur activation) ;
- les processus représentent un espace d'adressage mémoire ; ils peuvent contenir des composants de données, des sous-programmes ou des tâches ;
- les sous-programmes : un composant de ce type représente une séquence d'exécution appelée avec des paramètres et n'a pas d'état propre (pas de données statiques

par exemple). Les sous-programmes, à la manière des tâches, peuvent être organisés en groupes représentant ainsi des bibliothèques ;

- les données représentent des données statiques. Des composants peuvent partager la même ressource, leur exclusion mutuelle faisant alors partie des exigences.

La représentation graphique de chacun de ces composants est présentée sur la figure 11.2.



Figure 11.2 – Syntaxe graphique des composants logiciels AADL

Composants matériels. Les composants matériels d'une spécification AADL permettent de préciser les éléments sur lesquels l'application sera déployée. On distingue :

- les processeurs, qui sont les unités matérielles permettant aux tâches de s'exécuter. Ils peuvent contenir et accéder à de la mémoire, communiquer avec des périphériques ou avec d'autres processeurs au travers des bus ;
- les mémoires représentent des éléments permettant d'enregistrer du code ou des données binaires. Il peut aussi bien s'agir d'un composant de RAM que d'un composant matériel plus complexe, comme un disque. Leurs attributs typiques sont le nombre et la taille de leur espace d'adressage ;
- les bus sont les éléments permettant d'échanger des flots de contrôle ou de données entre les processeurs, les mémoires et les périphériques. Typiquement, ce sont des canaux de communication auxquels sont associés des protocoles. Ils peuvent se connecter directement entre eux afin de créer des réseaux complexes ;
- les périphériques (*devices*) représentent des éléments particuliers du matériel, des éléments extérieurs au système, ou encore des éléments interagissant avec l'extérieur depuis le système. Ces périphériques peuvent avoir leur propre processeur, mémoire et composants logiciels, qui peuvent être spécifiés séparément.

A ces éléments purement matériels s'ajoutent les composants « virtuels », processeurs et bus. Ils représentent une vue abstraite de ces composants : un processeur virtuel représentant un sous-ensemble d'un processeur (partition ou cœur d'un processeur, un bus virtuel représentant un protocole au-dessus d'un bus physique).

La syntaxe graphique de chacun de ces composants est présentée sur la figure 11.3

Hiérarchisation via les systèmes. Lors du processus de modélisation, certaines décisions d'affectation peuvent rester en suspens. On peut ainsi connaître l'interface d'un composant sans pour autant connaître sa catégorie. Les composants « abstraits »

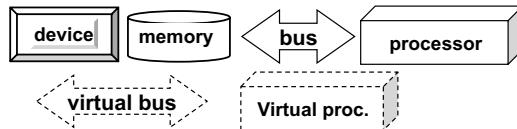


Figure 11.3 – Syntaxe graphique des composants matériels AADL

permettent de représenter ces composants. Ces composants ne peuvent être utilisés comme tels dans la configuration finale du système : ils doivent être raffinés en une catégorie concrète.

Les composants systèmes sont organisés selon une hiérarchie représentant la structure globale du modèle. Il s'agit d'un ensemble de composants logiciels, matériels, et d'autres systèmes interagissant ensemble. La syntaxe graphique d'un composant système est présentée sur la figure 11.4.



Figure 11.4 – Syntaxe graphique des composants systèmes d'une spécification AADL

11.4.2. *Connections entre composants*

Nous avons présenté les différents types de composants présents dans une spécification AADL. Ils interagissent grâce à leurs éléments d'interfaces.

Les caractéristiques (*features*) d'un composant spécifient la manière dont celui-ci interagit avec le reste du système au travers :

- des ports, qui sont les interfaces permettant aux composants d'échanger des événements ou des données. On les classe en trois catégories :
 - les ports de données (*data ports*),
 - les ports d'événements (*event ports*),
 - les ports de données et d'événements (*event data ports*).

Les ports ont une direction : un port de sortie sera connecté à un port d'entrée. On distingue les ports en entrée (*in*), sortie (*out*) et bidirectionnels (*inout*) ;

- des accès à des sous-programmes, modélisant les appels depuis des composants extérieurs d'une instance d'un sous-programme, par exemple un rendez-vous entre tâches ou un appel de procédure distante ;
- des paramètres, qui sont des valeurs de données qui sont passés en entrée (ou en sortie) des appels à des sous-programmes ;

- des accès à des données, spécifiant des accès à une variable partagée ;
- des accès à des bus qui représentent la connectivité physique des composants processeurs, mémoires et autres composants.

Certaines caractéristiques d'un composant peuvent être requises (*requires*) ou fournies (*provides*), permettant de définir des contrats entre composants.

Lors de la définition d'une implantation d'un composant, ces sous-composants sont connectés entre eux et à l'interface du composant père. Ce tissage de connexion permet de construire ainsi une configuration du système.

La figure 11.5 décrit la syntaxe graphique de ces différents éléments d'interface.



Figure 11.5 – Syntaxe graphique des éléments d'interface d'une spécification AADL

11.4.3. *Eléments de langage : attributs*

Les attributs AADL, tels que présentés dans la deuxième version du standard [SAE 09], se divisent en plusieurs catégories :

- les attributs de déploiement : composants liés à un processeur, une mémoire ;
- les attributs liés aux éléments de communication : tailles de files de messages, politiques de gestion de files, priorité des messages, traitement des événements entrants, politique de diffusion de messages ;
- les attributs liés aux éléments de stockage : taille mémoire, tas, pile ;
- les attributs de programmation : appels de procédures, priorité ;
- les attributs liés aux tâches : priorité, pire temps d'exécution ;
- les attributs temporels : dates d'échéances, délais de communication.

Le jeu d'attributs d'AADL est riche, et a été défini afin de permettre de nombreuses analyses. Nous présentons une classification non exhaustive de ces attributs, et les relions à des analyses possibles.

Attributs de déploiement. Ces attributs permettent de spécifier les contraintes de liaison qu'il peut y avoir entre un composant logiciel et un composant matériel (ou une catégorie de matériel).

Pour chaque attribut considéré, le standard définit trois variations autour d'un mot-clé, que nous notons χ :

- `Allowed_ χ _Binding_Class : permet de spécifier sous forme d'ensemble les classes d'éléments pouvant être liées au composant logiciel ;`
- `Allowed_ χ _Binding : ensemble de composants matériels pouvant être liés au composant logiciel ;`
- `Actual_ χ _Binding : désigne l'élément matériel lié au composant logiciel pour une configuration.`

Les mots-clés effectivement pris en compte dans le standard sont : `Processor`, `Memory` : `Connection` `Subprogram_Call`.

Outre ces attributs, se trouvent aussi ceux :

- indiquant si des composants sont déployés et liés sur le même matériel (regroupement logique des composants) ;
- spécifiant la qualité de service attendue par les connexions ;
- indiquant les politiques sont mises en œuvre (déclenchement pour les processeurs, mode de lecture pour la mémoire, ordonnancement des tâches) ;
- exprimant des limites, comme le nombre de tâches maximal géré par un processeur, ainsi que les bornes des priorités que l'on peut leur attacher.

Les attributs relatifs au déploiement peuvent être exploités de différentes façons pour l'analyse du système :

- validation de l'adéquation des composants déployés par rapport au matériel sous-jacent (respect des contraintes exprimées) ;
- validation de la sécurité du système : flots de communications dans le système ;
- informations pour architecturer un modèle formel (interactions entre composants logiciels et composants de ressources, partagées ou non).

Attributs liés aux communication. Ils permettent de caractériser une communication qui a lieu entre différents composants. On distingue :

- des politiques de qualité de service :
 - liées à l'émission de message (`Fan_Out_Policy`) : lorsqu'un message est envoyé, il peut être diffusé à tout le monde (*broadcast*), choisir un destinataire à tour de rôle (*round robin*), ou encore envoyer le message à la demande,

- liées à l'arrivée de nouveaux messages dans la file : en cas de file pleine (*Overflow_Handling_Protocol*) : il permet de supprimer de la file soit le message le plus récent, soit le plus ancien, ou encore de lever une erreur,
- liées à la paramétrisation de la connexion : taille de file (*Queue_Size*), ou encore le type de transmission (*push, pull*) ;
- des informations quant aux délais, taux d'envoi ou de réception :
- d'émission (*Output_Rate, Input_Rate*) : nombre de messages envoyés ou reçus par seconde (ou par déclenchement de tâche); (*Output_Time, Input_Time*) : moment où les messages sont envoyés ou reçus (avant le calcul du tâche, après, pendant, à la date d'échéance),
- de transmission (*Transmission_Time*);
- des attributs caractérisant la latence du système (d'un bout à l'autre du système, pour les flots de données).

Ces attributs offrent la possibilité d'analyser les communications du système : sûreté des communications en termes de perte de message, de temps d'acheminement, respect des échéances.

Attributs relatifs aux composants de stockage ou de transfert. Il est possible de définir les éléments pour dimensionner des tâches : la taille du tas (*Source_Heap_Size*), la taille de la pile (*Source_Stack_Size*), la taille des données (*Source_Data_Size*).

Les tâches sont liées (au travers du processus les hébergeant) à une zone mémoire physique, pour laquelle on peut spécifier :

- les permissions d'accès en lecture et/ou écriture en mémoire ou sur le bus (*Acces_Right*) : lecture ou écriture seule, lecture et écriture, ou encore accès uniquement via des appels de méthodes spécifiques ;
- le temps d'accès physique à la mémoire en écriture (*Write_Time*) ;
- adresses délimitant l'espace mémoire.

Ces attributs permettent d'effectuer des analyses sur la sécurité des interactions entre composants, et de bon dimensionnement des zones mémoires. Ces vérifications sont mises en œuvre dans l'outil Ocarina.

Attributs de programmation. Ces attributs offrent la possibilité de spécifier le comportement des tâches sur la réception d'événements extérieurs, provoquant le déclenchement de leur exécution, ou encore de lié code source et sous-programmes AADL.

On appelle point d'entrée (*entrypoint*) une référence vers un sous-programme. Il peut s'agir d'un sous-programme AADL, d'une séquence de sous-programmes ou d'une chaîne de caractères référençant une fonction écrite dans un autre langage. Ces points

d'entrée sont attachés à des éléments particuliers du cycle de vie, sous forme de propriété `X_EntryPoint` où `X` correspond à :

- `Activate` : tout ce qui concerne le tâche lors de sa sélection pour un mode d'exécution ;
- `Compute` : concerne l'exécution d'un tâche une fois son déclenchement effectué. S'il s'agit de l'attribut d'un port, alors il peut être dissocié de l'attribut similaire du tâche possédant le port (déclenchement sur réception d'un événement par exemple) ;
- `Deactivate` : programmes susceptibles d'être exécutés lors de la dé-sélection du tâche d'un mode en cours ;
- `Finalize` : code exécuté par un tâche lorsque celui-ci termine son exécution ;
- `Initialize` : ensemble des codes désignés pour s'exécuter lors de l'initialisation d'un tâche dans le système ;
- `Recover` : code exécuté par le tâche dans le cas où celui-ci passe en mode de gestion d'erreur (*recovery*).

Ces propriétés complètent la configuration du système en liant comportement de l'architecture et implantation de sous-programmes.

Attributs liés aux tâches. Ces attributs permettent de spécifier des informations relatives aux composants actifs de la spécification : ils permettent de renseigner les politiques de déclenchement des tâches, la concurrence et le passage d'un mode AADL à un autre. Ceux-ci sont classés comme suit :

- les informations liées au déclenchement de tâches : `Dispatch_Protocol` (apéridique, sporadique, périodique, etc.), `Dispatch_Trigger` (spécifie la liste des ports d'entrées susceptibles de déclencher le tâche sur la réception d'un message) ;
- les informations d'ordonnancement : `POSIX_Scheduling_Policy` : indique si le système est ordonné avec une politique FIFO, RR (Round Robin – à tourniquet), ou autre. En complément, viennent les attributs `Priority`, `Criticality`, `Urgency` et `Time_Slot`. Un attribut supplémentaire permet d'indiquer quelle politique appliquer pour l'exclusion mutuelle (`Concurrency_Control_Protocol`) ;
- les informations liées aux politiques de traitement des messages reçus : `Dequeue_Protocol` (traite un, plusieurs ou tous les messages).

Ces informations permettent d'analyser le comportement de l'application, les bornes sur la taille des files de messages ou encore l'ordonnancement du système.

Attributs temporels. La dernière catégorie d'attributs à présenter est celle des attributs spécifiant les temps d'exécutions liés aux composants comme les tâches, les périphériques ou encore l'exécutif sous-jacent.

On peut ainsi spécifier l'échéance (*deadline*) et le temps d'exécution (un intervalle de temps définissant les bornes de ce temps) pour les différents états d'une tâche ou d'un sous-programme : activation, initialisation, reconfiguration, etc.

D'autres attributs viennent compléter la description temporelle du système :

- ceux concernant le temps de chargement d'un tâche dans le système, durée d'un changement de contexte, latence sur les communications, etc. ;
- *Period*, qui donne le délai minimal entre deux déclenchements d'un tâche ;
- ceux qui concernent la notion de temps au sein de la plate-forme d'exécution : le *Jitter* de l'horloge matérielle, le temps pour l'exécutif d'effectuer un changement de contexte pour les processus, le temps mis pour effectuer un changement de contexte pour les tâches d'un même processus, etc.

Ces attributs permettent d'effectuer de nombreuses analyses temporelles sur l'ensemble du système, voire même d'affiner des analyses comportementales en éliminant, à l'aide des contraintes temporelles qu'ils expriment.

11.4.4. Eléments de langage : extensions et raffinements

AADL supporte la modélisation incrémentale de systèmes par raffinements et extensions successifs.

Le mot-clé «*extends*» permet d'indiquer qu'un composant (type ou implantation) étend un autre composant. Dans ce cas, les propriétés, interfaces et modes du composant père sont hérités. Il est alors possible de compléter l'interface du composant, surcharger certaines propriétés.

Dans certains cas, il est aussi nécessaire de surcharger un élément existant. AADL définit le mot-clé «*refined to*» qui permet d'indiquer qu'un élément (interface ou sous-composant) est raffiné vers un type plus concret.

11.5. Annexes à AADL

La section précédente a présenté le cœur d'AADLv2. Mais ce langage est prévu pour être extensible afin de m'enrichir des concepts complémentaires permettant de nouveaux types d'analyse. Le standard autorise donc la définition d'«annexes». Deux types d'annexes sont définis :

- les documents annexes définissent des guides de modélisation, des ensembles de propriétés complémentaires, parmi lesquels on peut citer l'annexe de modélisation des données, ou l'annexe ARINC653 pour modéliser des systèmes avioniques conformes aux patrons de conception de l'avionique modulaire intégrée (IMA) ;

– les langages annexes permettent d'adoindre à un modèle AADL une description complémentaire dans un autre langage pour compléter la description. Il peut s'agir d'une description du comportement, de la propagation des erreurs dans le système.

L'ensemble de ces annexes sont publiés par le comité de standardisation [SAE 11]. Un outil est libre de créer ses propres annexes, spécifiques. Le standard indique qu'une annexe ne peut entrer en conflit avec le standard, et peut être ignorée. Cette limitation permet de ne pas dénaturer la sémantique du standard.

Nous présentons par l'annexe de modélisation des données qui nous sera utile pour modéliser notre cas d'étude.

11.5.1. L'annexe de modélisation des données

AADL permet de définir des types de composants dont la catégorie correspond à une donnée. Les propriétés standards du langage fournissent quelques éléments quant à l'utilisation mémoire ou les protocoles d'accès concurrent à ces données. En revanche, rien n'est indiqué concernant la représentation de ces données : type simple ou composé, unité implicite, représentation mémoire, etc.

Le but de l'annexe de modélisation des données, publiée dans le document [SAE 11] répond à ces questions en définissant un ensemble de propriétés et des règles de composition permettant de définir précisément la nature des données utilisés par un système, que ce soit en paramètre d'un sous-programme ou les ports des composants.

Le jeu de propriétés `Data_Model` définit quinze propriétés additionnelles, nous en présentons quelques-unes :

- `Data_Representation` indique le type de base d'une donnée (entier, tableau, booléen, etc.). Cette propriété autorise l'emploi d'autres propriétés, en fonction de la représentation retenue ;
- `Data_Digits` et `Data_Scale` permettent d'indiquer la précision d'un nombre en virgule fixe; tandis que `Number_Representation` ou `IEEE754_Representation` permettent d'indiquer la résolution d'un nombre ;
- `Initial_Value` permet d'indiquer la valeur initiale d'une donnée ;
- `Integer_Range` (respectivement `Real_Range`) permet d'indiquer l'intervalle de valeur admisible pour un entier (respectivement un flottant) ;
- `Measurement_Unit` permet d'indiquer l'unité associé à une valeur.

Ces propriétés additionnelles permettent de raffiner l'analyse de compatibilité des interfaces, par exemple pour l'échange et le traitement de grandeurs physiques ou de garantir la précision des calculs dans une chaîne de capteurs/calculateurs. L'exemple

suivant montre comment modéliser une variable contenant la valeur de l'impédance d'une sonde du pacemaker, exprimée en ohm.

```

1 package Pacemaker_Data
2   public
3     with Data_Model;
4
5     data Impedance_Lead -- Impedance of a lead
6       properties
7         Data_Model::Data_Representation => Integer;
8         Data_Model::Initial_Value => «\,85\,»;
9         Data_Model::Measurement_Unit => «\,ohm\,»;
10        end Impedance_Lead;
11      end Pacemaker_Data;

```

11.6. Analyses de modèles AADL

AADL est un langage de spécification qui se prête particulièrement bien à des analyses variées pour les systèmes temps réel.

Dans une ingénierie classique dirigée par les modèles, l'ingénieur est en mesure, très rapidement et très facilement, de spécifier et d'annoter différentes vues de son système, séparant ses préoccupations. En procédant ainsi, lorsque viennent les étapes de validation et de vérification, l'ingénieur doit alors récupérer les informations disséminées dans toutes les vues de son système. Elles ne sont pas nécessairement orthogonales et peuvent être utiles pour effectuer plusieurs types d'analyses : bien souvent, il ne peut se contenter d'utiliser une vue particulière de son système, mais a besoin d'une composition de vues ou un sous-ensemble de celles-ci.

Se pose alors la question suivante : une fois le modèle utile produit, comment et pour quel(s) type(s) d'analyse(s) l'utiliser ? Avec quels outils ? Nous avons commencé à répondre à cette question à la section précédente. Pour compléter ces résultats, nous allons expliquer comment il est possible de centrer l'ingénierie de tels systèmes non plus sur les modèles-mêmes, mais sur les analyses que l'on souhaite effectuer, ce qui est aussi un des buts d'AADL.

Nous avons présenté les différentes catégories d'attributs AADL dans l'optique de les classer afin d'en faciliter l'analyse formelle. Nous allons indiquer, pour chacun de ces ensembles d'attributs, s'ils contiennent des informations permettant de caractériser :

- des propriétés structurelles ;
- des propriétés qualitatives ;
- des propriétés quantitatives.

Ceci est résumé dans le tableau 11.1.

Propriétés structurelles. Elles sont liées à la structure du système, comme :

Impact Attributs AADL	Structurel	Qualitatif	Quantitatif
Déploiement	×		×
Communication	×	×	×
Programmation		×	×
Tâches	×	×	×
Temps			×

Tableau 11.1 – Impact des attributs AADL sur les propriétés d’analyse

- les connexions et la cohérence entre les interfaces des composants du système ;
- les invariants à maintenir dans le système ;
- l’analyse de défaillances *Fault-tree analysis* (dépendance entre les composants quand l’un deux tombe en panne), de latence bout en bout de flots d’informations, de ressources énergétiques, etc.

La plupart de ces propriétés doivent être établies très tôt dans le processus de développement, souvent à faible granularité. Elles peuvent être raffinées ou enrichies quand la conception du système évolue.

Propriétés qualitatives.

Elles sont relatives au comportement du système, c’est-à-dire quant à son ordonnancement, la détection de famine, d’interblocage ou encore les liens de causalité entre composants.

Pour traiter de telles propriétés, le comportement du système doit être défini. Elles sont traditionnellement décrites plus tard dans le processus de développement, quand les informations relatives au comportement des composants deviennent accessibles grâce aux attributs de programmation ou encore l’annexe comportementale du langage.

Propriétés quantitatives.

Elles sont utilisées pour évaluer les performances du système ou pour évaluer son comportement selon des critères probabilistiques ou de temps d’exécution. Pour établir ce type de propriétés, des informations sur les temps d’exécution sont requises.

AADL est un langage de description d’architecture standardisé qui a prouvé qu’il était utilisable pour procéder à différentes analyses sur les systèmes temps réel : de nombreux travaux exploitant AADL dans cette optique ont été réalisés autour de BIP [CHK 09], de TLA+ [ROL 09], d’UPPAAL [PON 07], de LOTOS [HAM 07], de Lustre [JAH 07], de Cheddar [SIN 07] ou d’Archeopteryx [ALE 09]. Ces travaux

passés nous permettent de tirer une première classification de la manière dont AADL est utilisé lors des phases d'analyses d'un système.

Le tableau de la figure 11.2 présente, pour chacun des éléments d'un modèle AADL un récapitulatif des informations ou des facilités qu'ils apportent selon le type d'analyses visées. Les colonnes catégorisent différents types d'analyses qu'il est possible d'effectuer sur un système. Les lignes sont divisées en deux catégories :

- les attributs AADL : pour chaque élément AADL, le tableau indique sur quel type d'analyse la valeur de l'attribut aura un impact ;
- les méthodes d'analyse : le tableau indique quels types d'analyses sont susceptibles d'être effectués avec chacune d'entre elles.

Nous sommes maintenant en mesure de répondre aux questions suivantes :

- quel type de validation ou de vérification souhaite-t-on effectuer ?
- comment peut-on l'effectuer ?
- quelles sont les informations qui lui sont utiles ?

Il est également possible de naviguer dans le tableau de différentes façons : pour quelles méthodes d'analyse tel attribut sera-t-il utile ? Que peut-on vérifier ou valider si l'on souhaite utiliser une technique particulière, et que l'on souhaite se focaliser sur certains attributs ?

Ce tableau est une brique essentielle dans un processus de vérification et de validation formelle autour d'un langage de description d'architecture comme AADL.

11.7. Conclusion

Dans ce chapitre, nous avons présenté les concepts fondamentaux d'AADL2. Ce langage de description d'architecture a été défini pour permettre une description précise d'un système embarqué temps réel.

AADL a une double représentation textuelle et graphique que le concepteur peut manipuler, et une représentation XML pour aider à la conception d'outils d'analyse. Le langage a été conçu en se basant sur les concepts de base des systèmes embarqués (notion de tâches, sous-programme, périphériques, etc.) ; tout en offrant des mécanismes de modularité, encapsulation et une forme simple d'héritage. Par construction, ces concepts se projettent facilement vers des outils d'analyse ; et de nombreuses initiatives en ce sens ont vu le jour, permettant de couvrir l'analyse d'ordonnancement, de sécurité, ressources mémoire, la simulation ou encore la génération de code.

Nous présentons plusieurs de ces démarches dans les chapitres suivants, illustrées par notre cas d'étude du pacemaker.

	Cohérence d'interfaces	Invariants Système	Arbre de défaillance	Ordonnancement	Vivacité	Causalité Interblocages	Analyse de performances
Eléments AADL							
Déploiement	×		×			×	×
Temps		×		×			×
Communication	×					×	×
Programmation	×	×	×			×	
Tâches		×	×	×	×	×	×
Méthodes d'analyse							
Simulation				×			
Analyse sémantique	×			×			×
Vérification de type	×						
Preuve de théorème	×	×	×		×	×	
<i>Model-Checking</i> ...temporel ...stochastique	×	×		×	×	×	×

Tableau 11.2 – Croisement des attributs et de leur impact en termes d'analyse

11.8. Bibliographie

- [ALE 09] ALETI A., BJORNANDER S., GRUNSKE L., MEEDENIYA I., « ArcheOpterix : An extendable tool for architecture optimization of AADL models », Los Alamitos, CA, USA, IEEE Computer Society, p. 61-71, 2009.
- [ari07] ARINC 653, Rapport, Aeronautical Radio Incorporated, 2007.
- [CHK 09] CHKOURI M. Y., ROBERT A., BOZGA M., SIFAKIS J., « Translating AADL into BIP - Application to the Verification of Real-Time Systems », p. 5-19, Springer-Verlag, 2009.
- [CMU 09] CMU/SEI, Open Source AADL Tool Environment (OSATEv2), Rapport, CMU/-SEI, 2009.
- [FEI 00] FEILER P. H., LEWIS B., VESTAL S., Improving Predictability in Embedded Real-Time Systems, Rapport n° CMU/SEI-2000-SR-011, universit<E9> Carnegie Mellon, décembre 2000, la.sei.cmu.edu/publications.
- [HAM 07] HAMID I., NAJM E., « Real-Time Connectors for Deterministic Data-flow », *Proceedings of RTCSA'07*, p. 173-182, 2007.

- [JAH 07] JAHIER E., HALBWACHS N., RAYMOND P., NICOLLIN X., LESENS D., « Virtual execution of AADL models via a translation into synchronous programs », *EMSOFT*, p. 134-143, 2007.
- [LAS 09] LASNIER G., ZALILA B., PAUTET L., HUGUES J., « OCARINA : An Environment for AADL Models Analysis and Automatic Code Generation for High Integrity Applications », *Reliable Software Technologies'09 - Ada Europe*, vol. LNCS, Brest, France, p. 237-250, juin 2009.
- [MED 00] MEDVIDOVIC N., TAYLOR R. R., « A Classification and Comparison Framework for Software Architecture Description Languages », vol. 26, n° 1, p. 70-93, 2000.
- [PON 07] PONTISSO N., CHEMOUIL D., « Vérification formelle d'un modèle AADL à l'aide de l'outil UPPAAL », *Génie Logiciel*, vol. 80, p. 36-40, Hermès, mars 2007.
- [ROL 09] ROLLAND J.-F., Développement et validation d'architectures dynamiques, PhD thesis, Université de Toulouse, 2009.
- [SAE 09] SAE, Architecture Analysis and Design Language (AADL) AS-5506A, Rapport, The Engineering Society For Advancing Mobility Land Sea Air and Space, Aerospace Information Report, Version 2.0, Janvier 2009.
- [SAE 11] SAE, SAE Architecture Analysis and Design Language (AADL) Annex Volume 2, Rapport, Society of Automotive Engineers, 2011.
- [SCH 04] SCHMERL B., GARLAN D., « AcmeStudio : Supporting Style-Centered Architecture Development (Research Demonstration) », *Proceedings of the 26th International Conference on Software Engineering*, Edinbourg, Royaume-Uni, 23-28 mai 2004.
- [SIN 07] SINGHOFF F., « The Cheddar project : a free real time scheduling analyzer, beru.univ-brest.fr/~singhoff/cheddar/ », 2007.

Chapitre 12

Modélisation de l'étude de cas avec AADL

12.1. Introduction

Comme décrit au chapitre précédent, un modèle AADL fournit l'architecture logicielle et matérielle d'un système temps réel dans un objectif de conception et d'analyse de ce système. Dans ce chapitre, nous allons décrire le modèle AADL du pacemaker, qui sera utilisé dans les chapitres suivants pour illustrer son utilisabilité en termes d'analyse et de génération de code.

Outre la description du modèle lui-même, nous expliquons dans ce chapitre les choix de modélisation qui nous ont guidés vers l'architecture présentée. Ces choix s'appuient sur deux types de contraintes importantes dans toute démarche de modélisation : les caractéristiques du langage de modélisation choisi et les contraintes liées au domaine d'utilisation du système à modéliser. Les caractéristiques du langage AADL ont été données au chapitre précédent, alors que les principales caractéristiques du pacemaker ont été présentés au chapitre 3.

Lorsque cela sera nécessaire, nous rappellerons et détaillerons certaines des propriétés attendus d'un pacemaker. En effet, si le chapitre 3 détaille la spécification système du pacemaker, notre objectif de conception nous oblige ici à détailler son fonctionnement interne.

Précisons succinctement les sources d'information dont nous sommes partis pour aboutir à l'architecture présentée dans ce chapitre.

Chapitre rédigé par Etienne BORDE.

Sources d'information concernant le fonctionnement du pacemaker. Concernant les contraintes liées au domaine d'application, notre démarche de modélisation a pour principal point de départ la spécification système du pacemaker [Bos 07]. A partir de cette spécification et d'une documentation présentant plus en détail l'utilisation d'un pacemaker dans le monde médical [BAR 10], nous avons produit le modèle AADL décrit dans ce chapitre. Le premier document référencé détaille les exigences du pacemaker et sa spécification système, alors que le deuxième document décrit le fonctionnement du pacemaker en lien avec son domaine d'utilisation : le domaine médical. L'utilisation de ces deux documents nous a permis de comprendre d'une part les exigences liées à la conception du pacemaker, et d'autre part les phénomènes physiologiques auxquels sont rattachées ces exigences.

Sources d'information concernant le langage de modélisation. A des fins de conception, un langage de modélisation vise à définir un vocabulaire et des concepts qui permettent de représenter de façon non ambiguë les réponses apportées à des exigences.

Un tel langage impose alors des contraintes pour que l'architecture spécifiée soit cohérente et puisse être interprétée par différents outils. Ces contraintes ont été présentées au chapitre précédent, mais nous rappelons ici celles qui nous ont guidé dans notre démarche de modélisation. Un modèle AADL doit respecter des règles de structuration (composition, connexion, déploiement) précisés par le standard. Par exemple, un composant *system* peut être composé de sous-composants *system*, *process*, *processor*, et/ou *device*.

Compte tenu des règles de structuration d'un modèle AADL, nous avons procédé par itérations, qui ont toutes suivies les étapes du fil conducteur suivant :

- décomposition structurelle, déduite des contraintes structurelles imposées par la topologie des entités constitutives du pacemaker ;
- modélisation de l'architecture logicielle et du comportement de ses composants, déduite des contraintes de fonctionnement décrites dans la spécification système du pacemaker. Le fonctionnement issu de ces contraintes est détaillé dans le livre [BAR 10] ;
- modélisation du déploiement des fonctionnalités logicielles sur l'architecture matérielle, qui décrit l'allocation des entités logicielles sur les ressources matérielles.

Ainsi, pour chacune de ces étapes, la modélisation doit répondre aux contraintes de structuration du langage de modélisation choisi, tout en répondant le plus précisément possible aux exigences du système modélisé.

Le plan de ce chapitre suit les différentes étapes de modélisation de notre fil conducteur, en commençant par rappeler les contraintes du système, puis en délivrant notre modélisation des réponses à ces contraintes en AADL.

12.2. Rappels concernant la structure du pacemaker

Les premières contraintes de structuration du pacemaker proviennent de sa décomposition en trois sous-systèmes, qui jouent des rôles très différents, et doivent être implantés sur des sites de natures totalement différentes :

1) les électrodes (appelées *Leads* dans la spécification système) ont pour rôle de sonder l'activité naturelle du cœur et de stimuler son activité artificielle. A ce titre, les électrodes doivent être implantées directement dans le myocarde du patient ;

2) le générateur de pulsations (appelé *Pulse Generator* dans la spécification système) est responsable du contrôle de l'activité cardiaque, à la fois dans le but de superviser l'activité naturelle, et de déclencher si nécessaire l'activité artificielle. Le générateur de pulsations est donc connecté aux électrodes, mais il peut être implanté dans une zone inerte du corps humain afin de minimiser la gène ressentie par le patient ;

3) la station de supervision et de configuration (appelée *Device Controller-Monitor* dans la spécification système) permet au praticien de s'assurer du bon fonctionnement du pacemaker et de modifier ses paramètres de fonctionnement. Ainsi, l'entité de supervision et de configuration n'est pas nécessaire au fonctionnement nominal du générateur de pulsations. Elle n'est pas implantée dans le corps du patient, mais communique avec le générateur de pulsations par le biais d'un télémètre. Cela permet au praticien de visualiser les données enregistrées par le générateur de pulsations, et de modifier la configuration de ce dernier si nécessaire.

Cette première décomposition du pacemaker nous renseigne sur la nature des sous-systèmes : les électrodes sont de simples périphériques qui ne possèdent pas de ressources de calcul ; le générateur de pulsations est un système embarqué composé à la fois de composants matériels et logiciels ; l'entité de supervision et de configuration est un système informatique (non embarqué) dédié au personnel médical assurant l'implantation et le suivi du pacemaker.

En plus de la nature des sous-systèmes, la décomposition structurelle nous renseigne sur les connexions qui existent entre ces différents sous-systèmes. Reste à déterminer la nature des informations échangées à travers ces connexions. Entre les électrodes et le générateur de pulsations, les informations échangées sont de simples signaux électriques. La nature des informations échangées entre la station de configuration et le générateur de pulsations est plus variable et complexe : certaines informations correspondent à des données envoyées par le générateur de pulsations à la station de configuration, d'autres à des ordres de configuration ou requêtes de supervision envoyées depuis la station de configuration vers le générateur de pulsations.

La section suivante décrit le modèle AADL de ce premier niveau de décomposition.

12.3. Modélisation AADL de la structure du pacemaker

Nous présentons dans cette section la modélisation AADL du premier niveau de décomposition : la structuration système. Nous avons organisé cette présentation en deux sous-parties. D'une part, nous nous intéressons à la décomposition du système en un assemblage de sous-systèmes interconnectés ; d'autre part, nous représentons la structure physique du pacemaker, c'est-à-dire les ressources de calcul et de communication qui exécutent les fonctionnalités du pacemaker. Cette deuxième étape est un raffinement de la première : chaque ressource de calcul identifiée est un sous-composant de l'un des sous-systèmes du pacemaker.

12.3.1. Décomposition du système en sous-systèmes

Les contraintes structurelles que nous avons présentées à la section 12.2, ainsi que la spécification système du pacemaker [Bos 07], nous ont incités à modéliser la structure du pacemaker en un ensemble de trois composants AADL de la catégorie *system*. Ces sous-systèmes correspondent (i) aux électrodes, (ii) au générateur de pulsations et (iii) à la station de configuration/supervision.

A partir de cette décomposition et de la description de la nature des interactions entre ces sous-systèmes, nous avons ajouté les interfaces de communication entre les composants AADL correspondants.

Nous avons choisi de décrire les interfaces de communication entre le sous-système d'électrodes et le générateur de pulsations par le biais de ports d'événements AADL. En effet, un port d'événements AADL constitue l'extrémité d'un canal d'échange de signaux logiques entre deux entités : le contenu de l'information qui transite par ce port représente l'occurrence du signal correspondant. Par analogie, l'information qui transite entre les électrodes et le générateur de pulsations correspond à l'occurrence de signaux électriques provenant du cœur ou à destination du cœur.

Par conséquent, nous avons ajouté deux ports d'événements en entrée/sortie sur chacun de ces deux sous-systèmes : électrode et générateur de pulsations. L'un de ces ports sera connecté à l'électrode implantée dans l'oreillette, alors que l'autre sera connecté à l'électrode implantée dans le ventricule. L'utilisation de ports en entrée/sortie vient du fait que chaque électrode pourra jouer le rôle de sonde (elle est alors émettrice de signal : port en sortie du système « électrode ») ou de stimulateur du myocarde (elle reçoit l'ordre du générateur de pulsations : port en entrée du système « électrode »). Les caractéristiques du signal électrique correspondant aux électrodes (l'impédance par exemple) seront décrites sous la forme de propriétés associées aux ports des composants AADL représentant ces électrodes.

La modélisation des interfaces entre le sous-système de configuration et le générateur de pulsations nécessite davantage de travail, car la nature des interactions est plus

variée. Nous allons nous limiter à un sous-ensemble de ces interactions, en considérant que la complétude de ces définitions ne présente pas de complexité supplémentaire : il suffira d'étendre la modélisation proposé en s'appuyant sur les mêmes concepts que ceux que nous présentons.

Nous nous focalisons sur deux types d'interactions, à savoir celles qui permettent au praticien :

- 1) de visualiser l'état de fonctionnement du générateur de pulsations par l'interface de supervision. Nous nous focalisons ici sur les données mesurées et en particulier le niveau de batterie ;
- 2) de modifier la configuration du générateur de pulsations, soit par le biais d'un changement du mode de fonctionnement (défini au chapitre 3 : *O, A, V, D*), soit en modifiant les caractéristiques du générateur de pulsations, comme par exemple le rythme minimal de battement (défini au chapitre 3 : *RL*) ou le délai minimal entre un battement dans l'oreillette et un battement dans le ventricule (défini au chapitre 3 : *AV*).

Dans notre modélisation AADL, nous réunissons ces interfaces au sein de groupes d'interfaces AADL (*feature group*), ce qui permet de limiter le nombre de connexions entre les deux sous-systèmes considérés. En effet, nous n'aurons alors qu'une seule connexion entre ces groupes d'interfaces (*feature group*) contre une connexion par interface sans utiliser ce concept.

Pour modéliser les différentes interfaces du groupe, nous devons choisir parmi les types d'interfaces proposées par le langage AADL : *data port, event data port, data access, subprogram access, etc.* Ce choix est guidé par la nature de l'interaction (échange de données ou non, données partagées entre différentes entités, l'interaction est-elle critique pour le bon fonctionnement du pacemaker et en particulier pour le respect des caractéristiques temporelles) :

- la mesure des paramètres de fonctionnement du générateur de pulsations est exécutée par le générateur mais initiée par l'entité de supervision. Nous allons modéliser cette interaction par le biais d'un accès à un sous-programme AADL (*subprogram access*). Ainsi, lorsque le praticien a besoin de connaître les paramètres de fonctionnement du générateur, la station de configuration/supervision sollicite cette interface. Ceci déclenche un traitement de mesure sur le générateur de pulsations, qui renvoie le résultat à l'entité manipulée par le praticien. Cet accès au sous-programme est fourni par le générateur (qui exécute le sous-programme correspondant) et requis par l'entité de supervision (qui déclenche la mesure) ;

- les changements de modes AADL sont déclenchés par l'intermédiaire de ports d'évènements. Nous ajoutons donc autant de ports d'évènements que de modes de fonctionnement que nous modélisons. Chaque port est identifié par un nom composé du préfixe *to_* puis du mode de fonctionnement. Ainsi, *to_M* sera utilisé pour identifier un évènement qui déclenche un changement de mode ayant pour cible le mode de fonctionnement « M » ;

– les interfaces de réglage des paramètres de configuration du générateur de pulsations sont quant à elles représentées par des ports de données, qui permettront de faire transiter les nouvelles valeurs de configuration depuis la station de configuration vers le générateur de pulsations. La modification de ces paramètres de fonctionnement n’entraîne pas de traitement particulier sur le générateur de pulsations (outre la modification d’une valeur de configuration). Ainsi il n’est pas utile d’associer un évènement (qui aurait déclenché l’exécution d’une tâche) à ce port de données.

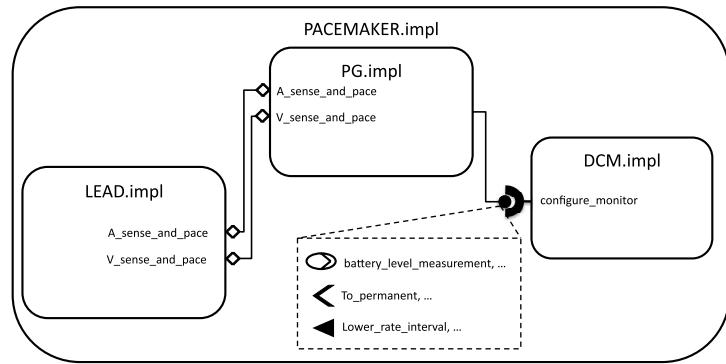


Figure 12.1 – Structure système du pacemaker en AADL

La figure 12.1 représente la structuration du pacemaker et de ses sous-systèmes, ainsi que les interfaces et connexions qui permettent à ces sous-systèmes d’interagir : le composant principal (*PACEMAKER.impl*) et composé de trois sous-composants (*LEAD.impl*, *PG.impl*, et *DCM.impl*) qui correspondent (respectivement) au sous-système d’électrodes, au sous-système de génération de pulsations, et à la station de configuration et de supervision. Conformément aux explications que nous avons données dans cette section :

- le générateur de pulsations est connecté aux électrodes par deux ports d’évènements en entrée-sortie correspondant à chacun des compartiments du cœur : *A_* pour *Atrial* (oreillette en français) et *V_* pour *Ventricle* (ventricule en français) ;
- le système de génération de pulsations est connecté au système de supervision et de configuration par le biais d’un groupe d’interfaces qui contient des accès à des sous-programmes (*battery_level_measure, etc.*) pour la mesure de paramètres de fonctionnement du générateur, des ports d’évènements pour les changements de modes (*to_Permanent, etc.*), et des ports de données pour modifier la configuration du générateur (*Lower_Rate_Limit, etc.*) .

La description de la structure du pacemaker, que nous avons présenté en section 12.2, fournit également la répartition de ces sous-systèmes en différentes ressources matérielles interconnectées. La sous-section suivante présente la modélisation AADL de la plateforme d'exécution correspondante.

12.3.2. Infrastructure d'exécution et de communication

La nature de la plate-forme matérielle qui permet d'exécuter les fonctionnalités du pacemaker dépend du sous-système considéré. Le sous-système des électrodes est constitué de simples périphériques matériels qui envoient un signal électrique au générateur de pulsations. Nous modéliserons donc chaque électrode par le biais d'un périphérique AADL (composant *device*).

Le pacemaker est constitué de deux électrodes, chacune étant placée dans un des compartiments du cœur. Par conséquent, nous aurons dans le modèle AADL deux périphériques : un par compartiment cardiaque. D'autre part, chacun de ces périphériques délivre au générateur de pulsations le signal électrique correspondant aux battements (naturels ou artificiels) du cœur. Par conséquent, les périphériques AADL posséderont chacun un port d'événements en entrée/sortie, connecté au port correspondant du système de génération de pulsations (*A_sense_and_pace* ou *V_sense_and_pace* sur la figure 12.1). Les périphériques correspondant aux électrodes sont alors connectés à un bus de communication qui sera lui-même connecté à la plate-forme d'exécution du générateur de pulsations. Ce lien représente le support matériel (fil électrique dans ce cas) qui achemine les signaux entre les électrodes et le générateur de pulsations. Ce dernier, ainsi que l'entité de configuration et de supervision du générateur, sont des calculateurs qui nécessitent une ressource matérielle capable d'exécuter leurs fonctionnalités. Nous modélisons donc la plate-forme de chacun de ces sous-systèmes par un processeur AADL (composant *processor*). Les deux processeurs ainsi modélisés sont connectés par un bus qui représente le télémètre, support des communications entre le générateur de pulsations et l'entité de configuration et supervision.

La figure 12.2 représente la plate-forme d'exécution du pacemaker. Comme le montre cette figure, les périphériques (*Atrial_Lead* et *Ventricle_Lead*) sont définis comme des sous-composants du système *LEAD.impl*. De la même façon, *PG_board* (processeur du générateur de pulsations) et *DCM_board* (processeur de l'entité de configuration et supervision) sont respectivement définis comme sous-composants des systèmes *PG.impl* et *DCM.impl*.

De plus, les ports d'événements des périphériques sont bien connectés aux ports d'événements du générateur de pulsations. Ces liens permettent de modéliser des interactions entre composants matériels et composants logiciels car ces deux domaines ne sont pas indépendants l'un de l'autre, et la modélisation de leurs dépendances ne peut

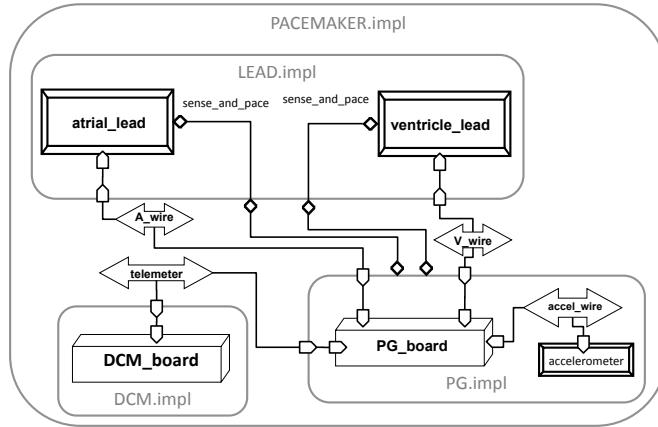


Figure 12.2 – Structure matérielle du pacemaker en AADL

se limiter à des relations d’allocation entre composants logiciels et composants matériels.

Enfin, les bus *A_wire*, *V_wire* et *Telemeter* représentent respectivement les canaux physiques de communication entre :

- l’électrode placée dans l’oreillette et le processeur du générateur de pulsations ;
- l’électrode placée dans le compartiment ventriculaire et le processeur du générateur de pulsations ;
- le processeur du générateur de pulsations et le processeur de l’entité de configuration et de supervision.

La plate-forme d’exécution ainsi décrite est prête à accueillir les fonctionnalités du pacemaker, et à faire interagir ses différents composants logiciels et matériels. Ainsi, nous continuons la modélisation AADL du pacemaker en suivant la deuxième étape de notre fil conducteur, présenté dans l’introduction de ce chapitre : la modélisation de l’architecture logicielle et du comportement des composants de cette architecture.

Pour continuer d’illustrer cette modélisation tout en restant synthétique, nous avons choisi de nous concentrer sur le modèle des fonctionnalités du générateur de pulsations. Cette restriction ne signifie pas que nous ignoreront les interactions du générateur de pulsations avec son environnement, mais simplement que nous ne modéliserons pas les composants logiciels de la station de configuration et supervision. Ce choix n’est pas tout à fait innocent : le générateur de pulsations est bien sûr l’élément central et critique du pacemaker puisqu’il coordonne les stimulations produites par le pacemaker.

Nous allons entrer dans une nouvelle phase de notre démarche de modélisation : il s'agit désormais de concevoir une application logicielle qui répond aux exigences système du pacemaker. Or, comme nous l'avons expliqué en introduction de ce chapitre, cela nécessite avant tout de comprendre les liens entre ces exigences et le domaine d'application du pacemaker et par la même le détail le fonctionnement du cœur et de savoir précisément à quel type d'anomalies cardiaques répond un mode de fonctionnement donné. La finalité de cet apprentissage est de mieux comprendre le fonctionnement attendu du générateur de pulsations, et ainsi de concevoir précisément l'application. Nous ne prétendons pas ici faire une description exhaustive, et le lecteur intéressé pourra se référer au manuel [BAR 10] pour plus d'explications. Cependant, avant de présenter notre modélisation, nous devons expliquer un certain nombre de fonctionnalités fournies par le générateur de pulsations.

12.4. Rappels concernant le fonctionnement du pacemaker

Nous faisons dans cette section un bref rappel concernant les fonctionnalités du pacemaker. Nous commençons par rappeler et préciser la décomposition du fonctionnement du pacemaker en modes opérationnels, puis nous décrirons les fonctionnalités du pacemaker associées à un sous-ensemble de ces modes. Nous avons choisi de nous limiter à un tel sous-ensemble car la modélisation exhaustive du pacemaker serait difficile à décrire dans ce seul chapitre.

12.4.1. *Les modes de fonctionnement du pacemaker*

Le fonctionnement du pacemaker (ou plus précisément du générateur de pulsations) se décompose en modes de fonctionnement. Un mode de fonctionnement permet d'identifier de façon synthétique un ensemble de fonctionnalités que doit fournir un système lorsqu'il est utilisé dans ce mode de fonctionnement. Le pacemaker présente un nombre important de modes de fonctionnement, organisés de façon hiérarchique : les cinq modes *permanent*, *temporary*, *pace-now*, *magnet*, et *power-on-reset* sont les modes principaux du pacemaker. Dans chacun de ces modes, le pacemaker rend un sous-ensemble des fonctionnalités identifiées par le biais d'un sous-mode de fonctionnement. Par exemple, dans le mode *permanent*, il existe dix-neuf sous-modes de fonctionnement. Nous allons nous intéresser plus particulièrement au mode de fonctionnement *permanent* car il constitue le mode de fonctionnement nominal du générateur de pulsations.

12.4.2. *Les sous-modes de fonctionnement du pacemaker*

Dans le mode *permanent*, le fonctionnement du pacemaker est décomposé en dix-neuf sous-modes de fonctionnement en réponse aux critères suivants :

- 1) quels sont les compartiments cardiaques stimulés (aucun, l'oreillette, le ventricule, les deux) ?
- 2) quels sont les compartiments cardiaques sondés (aucun, l'oreillette, le ventricule, les deux) ?
- 3) quel type de réponse est apporté à une détection de battement (aucune, immédiate, inhibée, suivie) ?
- 4) le rythme de battement est-il constant, ou régulé en fonction des activités du patient, détectées par un accéléromètre ?

1) et 2) Capteurs et actionneurs. L'explication des deux premiers critères est triviale, il s'agit principalement de considérer quels sont les capteurs et actionneurs auxquels a accès le générateur de pulsations pour réguler l'activité cardiaque.

3) Type de réponse. Le troisième critère correspond à différents comportements du pacemaker :

- lorsque le type de réponse aux mesures est « aucune », les stimulations sont provoquées sans aucune synchronisation avec la détection de battements naturels ;
- lorsque le type de réponse aux mesures est « immédiate », la détection d'un battement dans un compartiment entraîne immédiatement une stimulation du compartiment correspondant ;
- lorsque le type de réponse aux mesures est « inhibée », la détection d'un battement dans un compartiment entraîne l'inhibition d'une stimulation en attente pour ce compartiment ;
- lorsque le type de réponse aux mesures est « suivie », un battement dans l'oreillette doit être suivi de la stimulation du ventricule après un délai fixe appelé *AV_delay* (délai entre un battement dans l'oreillette et un battement dans le ventricule), sauf si entre temps, un battement a été détecté dans le ventricule.

4) Fréquence de régulation. Le dernier critère permet de modifier la façon dont est déterminé le rythme de régulation de l'activité cardiaque (avec ou sans considération pour les mouvements du patients).

Les différents sous-modes du pacemaker sont alors identifiés dans la spécification système par le biais d'une suite de quatre lettres qui représentent une à une la réponse aux trois premiers critères présentés ci-dessus. Pour les critères 1) et 2), les lettres possibles sont O pour aucun (*None* en anglais), A pour l'oreillette (*Artium* en anglais), V pour le ventricule (*Ventricle* en anglais) ou D pour les deux (*Dual* en anglais). Pour le critère 3, nous avons O pour aucune (*None*), T pour immédiate (*Triggered*), I pour inhibée (*Inhibited*), ou D pour suivie (*Tracked*). Enfin, la lettre R (*Rate*) peut être ajoutée si le rythme de régulation cardiaque dépend des mouvements du patient, détectés par un accéléromètre.

Par exemple, VDDR signifie que seul le ventricule est stimulé ; que les deux compartiments sont sondés ; que le type de réponse est « suivie » ; et que les mouvements du patient sont pris en compte pour déterminer le rythme de battements.

Les types de réponses (critère 3) du pacemaker dépend principalement de la maladie cardiaque traitée. Insuffisance cardiaque (réponse « immédiate »), valvulopathie (réponse « inhibée » ou « suivie ») ou arythmie (réponse « suivie »), vont guider la configuration du pacemaker dans l'un ou l'autre de ces modes de fonctionnement.

Il est évident que le comportement du générateur de pulsations sera très différent dans chacun de ces modes. De plus, la spécification ainsi proposée est trop imprécise pour permettre une conception détaillée de ces fonctionnalités. Pour être en mesure de concevoir ces fonctionnalités de façon détaillée, nous devons préciser le comportement du générateur de pulsations dans chacun de ces modes de fonctionnement. Nous avons choisi de nous intéresser plus particulièrement à quatre de ces modes (étudier les 19 modes dans le cadre de ce chapitre ne serait pas possible).

12.4.3. Quelques fonctionnalités du pacemaker

La fonctionnalité principale du pacemaker est de générer des pulsations cardiaques, ce qui est réalisé par un logiciel embarqué. Ce logiciel est un régulateur de pulsations, c'est-à-dire une boucle de contrôle qui, à partir des mesures de battements cardiaques, et en fonction du mode de fonctionnement sélectionné, délivre des stimuli vers les muscles du cœur (soit dans le compartiment ventriculaire, soit dans l'oreillette, soit dans les deux).

Comme nous l'avons expliqué, le pacemaker offre un nombre important de modes de fonctionnement. Dans un objectif de synthèse, nous choisissons de nous focaliser sur le mode permanent (mode nominal du générateur), et sur les sous-modes de fonctionnement suivants : DOOR, AAI, VVT, et VDD. Cela permet de représenter différents types de relations entrées/sorties, et de représenter tous les types de réponse du générateur de pulsations (O, I, T, et D).

Nous décrivons ci-dessous les fonctionnalités associées à chacun des ces modes de fonctionnement.

Permanent/DOOR. Dans ce mode, les deux chambres (oreillette et ventricule) sont stimulées, aucune d'entre elles n'est sondée, et le type de réponse aux mesures est « aucune ». Avec ce type de réponse, le pacemaker bat à un rythme indépendant des battements intrinsèques du cœur du patient. Un battement dans l'oreillette est séparé d'un battement dans le ventricule par un délai déterminé par la période oreillette-ventricule. Le rythme de battement est déterminé grâce à des données sur l'activité physique du patient, fournies par un accéléromètre.

Ces caractéristiques sont résumées par trois exigences temporelles, qui doivent être respectées par le pacemaker :

- le délais *AV*, délai minimal entre un battement dans l'oreillette et un battement dans le ventricule ;
- la limite *LRL*, qui impose un rythme minimal (ou délai maximal) entre deux cycles de battements ;
- la limite *URL*, qui impose un rythme maximal (ou délai minimal) entre deux cycles de battements.

Ce type de réponse fut le type de réponse principale des premières générations de pacemakers. Dans les pacemakers plus modernes, ce type de réponse est obsolète et ne sert qu'à tester le pacemaker lorsqu'un aimant spécial est placé au-dessus du pacemaker (mode principal *magnet*). Ce mode est également disponible dans le mode *permanent* de nombreux pacemakers pour assurer la compatibilité ascendante des fonctionnalités.

Puisque le pacemaker sollicite le cœur indépendamment des battements intrinsèques de celui-ci, il y a donc concurrence entre les battements naturels et les battements artificiels induits par le pacemaker. Cependant, les sollicitations ne se transformeront en battements que s'ils sont émis en dehors de la période réfractaire absolue du compartiment concerné. De plus, le risque de fibrillation d'un compartiment cardiaque du fait d'un battement émis dans la période de vulnérabilité de ce compartiment est très faible, et associé à des facteurs de risques bien identifiables médicalement.

Pour réduire encore davantage ce risque, les types de réponses implantées sur les nouvelles générations de pacemaker visent à supprimer la concurrence au niveau des battements cardiaques, ce que propose le type de réponse *Inhibited*.

Permanent/AAI. Dans ce mode, seule l'oreillette est stimulée et sondée. Le type de réponses aux mesures est « inhibée », ce qui signifie que le pacemaker envoie des stimuli à l'oreillette de façon périodique, sauf si un battement naturel est détecté avant l'échéance d'une période. Dans ce cas, l'horloge interne du générateur de pulsations est réinitialisée et le stimuli ne sera émis par le pacemaker que lorsque la période suivante (fixée par la limite *LRL*) sera atteinte et à condition qu'aucune activité naturelle n'ait été détectée entre temps.

On peut noter ici que la détection des battements intrinsèques de l'oreillette ne commence qu'après un délai fixe correspondant à la période réfractaire de l'oreillette. Ceci afin de ne pas considérer les résidus électriques d'un battement comme de véritables nouveaux battements.

Ce mode de fonctionnement est donc caractérisé par trois exigences temporelles : *LRL* et *URL*, qui ont le même rôle que précédemment, et *ARP*, qui représente la

période réfractaire de l'oreillette. Une fois qu'un battement est détecté ou déclenché dans l'oreillette, de nouvelles détections sont ignorées jusqu'à ce que ce délai soit passé.

Dans ce mode, il n'y a plus de concurrence des battements naturels et artificiels. En revanche, il est important de noter que ce mode est utilisé sur des patients dont la conduction entre l'oreillette et le ventricule est normal. Ainsi, la stimulation de l'oreillette provoquera une dépolarisation progressive du cœur et entraînera un battement ventriculaire après le délai oreillette-ventricule (AV) naturel. Le délai AV n'est donc pas considéré dans la conception logicielle de ce mode de fonctionnement.

Permanent/VVT.

Dans le mode VVT, seul le compartiment ventriculaire est sondé et stimulé. Plus précisément, le phénomène mesuré est la dépolarisation de l'interface entre l'oreillette et le ventricule, représentative d'un battement de l'oreillette suivi de la période oreillette-ventricule.

Le type de réponse immédiate impose que le compartiment ventriculaire soit stimulé dès que la dépolarisation de l'interface oreillette-ventricule est détectée. Cependant, le nom de ce type de réponses est trompeur car il faut également respecter les contraintes temporelles suivantes :

- 1) le rythme minimal de battement (LRL pour *Lower_Rate_Limit*), soit une période au bout de laquelle le compartiment ventriculaire sera stimulé ;
- 2) la période réfractaire du ventricule (VRP pour *Ventricle_Refactory_Period*), pendant laquelle les mesures sont ignorées ;
- 3) le rythme maximal de battement (URL pour *Upper_Rate_Limit*), ne considérant les mesures que si elles sont séparées par une période minimale.

Ce mode de fonctionnement s'adresse également aux patients dont la conductivité oreillette-ventricule est normale. Le but est ici d'assurer la stimulation plutôt que l'inhibition. Ce mode est rarement utilisé aujourd'hui, à la faveur du mode VVI.

Permanent/VDD.

Dans ce mode, les deux compartiments cardiaques sont sondés, et seul le ventricule est stimulé. Lorsque le pacemaker détecte un battement dans l'oreillette, le générateur de pulsations attend un délai identifié dans la spécification système par la variable AV et qui correspond à l'intervalle oreillette-ventricule. A l'issue de ce délai, deux situations se présentent. Soit le pacemaker a, entre temps, détecté un battement naturel dans le compartiment ventriculaire, auquel cas le générateur de pulsations ne fait rien, soit dans le cas contraire, le générateur de pulsations envoie un stimuli vers le ventricule.

Ce mode de fonctionnement est caractérisé par quatre intervalles de temps :

- l'intervalle oreillette-ventricule (AV) ;
- l'intervalle minimal entre deux pulsations (représenté par URL) ;
- l'intervalle maximal entre deux pulsations (représenté par LRL) ;
- la période réfractaire post-ventriculaire de l'oreillette (PVRP) qui permet d'éviter de considérer comme un battement de l'oreillette un résidu électrique provenant d'un battement ventriculaire.

Maintenant que nous avons détaillé ces fonctionnalités, nous allons détailler leur modélisation en AADL.

12.5. Modélisation AADL de l'architecture logicielle du générateur de pulsations

La modélisation des fonctionnalités du générateur de pulsations utilise les composants AADL suivants : un processus (*process*), qui représente le programme déployé sur la plate-forme d'exécution du générateur, et plusieurs tâches (*thread*), qui représentent les fonctionnalités principales du programme, c'est-à-dire la boucle de contrôle, la configuration et la supervision du système.

12.5.1. Modélisation AADL des modes de fonctionnement du générateur de pulsations

Dans le modèle AADL, nous énumérons l'ensemble des modes de fonctionnement du pacemaker. Nous aurons donc cinq modes principaux, auxquels sont associés un certain nombre de sous-modes : dix-neuf dans le mode *permanent*, quatre dans le mode *temporary*, un dans le mode *pace-now*, quatre dans le mode *magnet*, et un dans le mode *power-on-reset*. Chacun de ces modes de fonctionnement correspond à un sous-ensemble différent de fonctionnalités, que l'on peut représenter en AADL grâce à des variations sur l'architecture logicielle du générateur de pulsations. Il suffit pour cela d'associer à une entité du modèle l'ensemble des modes de fonctionnement dans lesquels cette entité reste accessible.

Pour représenter l'ensemble des modes dans un modèle AADL, plusieurs possibilités s'offrent à nous.

Commençons par une mauvaise solution, pourtant classique en informatique : aplatiser la structure hiérarchique. En effet, nous pourrions énumérer l'ensemble des modes (vingt-neuf au total) en les identifiant par concaténation du nom du mode principal et de la série de lettres identifiant le sous-mode. Nous aurions alors les modes suivants : *permanent_Off*, *permanent_DDDR*, *permanent_VDDR*, ..., *temporary_OVO*, *temporary_OAO*, ..., *pace-now_VVI*, *magnet_AOO*, *magnet_VOO*, ..., *power-on-reset_VVI*.

Dans un soucis de synthèse, nous n'avons pas écrit la liste exhaustive des modes de fonctionnement. Celle-ci peut facilement être déduite après lecture de la spécification système du pacemaker [Bos 07]. Pour représenter les variations de fonctionnalités fournies par le générateur de pulsations en fonction du mode de fonctionnement utilisé, il suffit alors d'associer l'ensemble des modes de fonctionnement à la tâche principale du générateur (composant *thread*), puis d'associer un comportement différent à cette tâche dans chacun des modes.

Nous avons adopté une autre solution, qui vise à conserver la structure hiérarchique des modes. Nous obtenons ainsi un modèle plus lisible, donc plus facile à maintenir. Pour cela, nous associons les cinq modes principaux du pacemaker au processus de contrôle (composant *process*). Nous ajoutons ensuite une implémentation de la tâche (composant *thread implementation*) par mode de fonctionnement, puis nous associons à chaque tâche les sous-modes de fonctionnement du mode principal que cette tâche représente. Enfin, nous associons un comportement différent à cette tâche dans chacun de ces sous-modes.

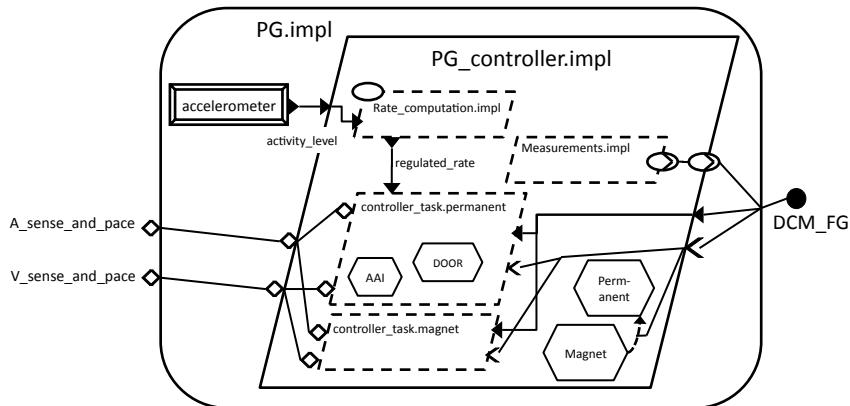


Figure 12.3 – Structure logicielle du pacemaker en AADL

La figure 12.3 représente l'architecture AADL du générateur de pulsations. Cette figure exhibe quatre parties importantes du fonctionnement du générateur de pulsations :

- la gestion des modes de fonctionnement : comme nous l'avons expliqué précédemment, nous avons choisi de représenter les modes principaux comme des modes du composant processus *PG_controller.impl*. Nous avons représenté deux de ces modes (*Permanent* et *Magnet*) sur la figure 12.3. Les ports d'événements du groupe d'interfaces (*DCM_FG*) connecté à la station de configuration pilotent les changements de modes ;

- la gestion des sous-modes de fonctionnement : les sous-modes de fonctionnement sont associés à chacune des tâches correspondant à un mode principal. Pour la tâche de régulation des battements dans le mode *Permanent* (*controller_task.permanent*), nous avons représenté deux de ces modes (AAI et DOOR) sur la figure 12.3. Le pilotage des changements de sous-modes se fait également par le biais du groupe d’interfaces (*DCM_FG*) connecté à la station de configuration ;
- la régulation des battements en fonction de l’activité physique du patient : un accéléromètre (*accelerometer*), connecté à une tâche périodique (*Rate_computation.impl*), fournit les données correspondant à l’activité physique du patient. La tâche *Rate_computation.impl* convertit ces données en période de battements que doit respecter la tâche de régulation (*controller_task.permanent*) ;
- la mesure des paramètres du pacemaker : une tâche *background* (*Measurement.impl*) est ajoutée au processus *PG_controller.impl*. Cette tâche, activé sur réception d’une requête de mesure par une interface d’accès à un sous-programme, permet de mesurer les paramètres de fonctionnement du générateur de pulsations ; et de les envoyer à la station de configuration et supervision.

Insistons ici sur le fait qu’un changement de modes peut altérer la structure de l’architecture logicielle. par exemple, la connexion logique entre le port en sortie de l’électrode placée dans l’oreillette et le port en entrée correspondant du générateur de pulsations ne sera accessible que dans les sous-modes du type XAXX et XDXX. Il s’agit ici d’une connexion/déconnexion logique, la non-accessibilité sera implantée de façon logicielle en ignorant les informations provenant de ce port. Il en est de même pour la connexion entre l’accéléromètre et le générateur de pulsations pour les modes XXXR ou XXX.

La suite de cette section détaille le comportement de la tâche de contrôle associée au mode permanent, dans les sous-modes auxquels nous avons choisi de nous intéresser. En effet, la variation du type de réponses associé à chaque sous-mode nécessite une modélisation plus fine du comportement de l’application logicielle.

12.5.2. Modélisation AADL des fonctionnalités du générateur de pulsations dans le mode permanent

Nous avons modélisé le fonctionnement de la boucle de contrôle du générateur de pulsations à l’aide d’une seule tâche associée au mode permanent. Cette tâche (*controller_task.permanent*) est représentée de façon incomplète sur la figure 12.3.

Dans cette sous-section, nous nous focalisons sur le fonctionnement de cette tâche et sur sa modélisation complète en AADL. Notons dès lors et déjà que le comportement associé à cette tâche sera différent dans chacun des sous-modes du mode permanent. Pour rester synthétique tout en couvrant un spectre de modélisation conséquent, nous

avons choisi de modéliser les sous-modes de fonctionnement suivants : DOOR, AAI, VDD et VVT.

Avant de présenter le comportement de la tâche de contrôle dans chacun de ces sous-modes, complétons sa description en y ajoutant les éléments dont nous aurons besoin dans la suite de cette section.

Nous pouvons tout d'abord préciser le type de protocole de déclenchement (propriété *Dispatch_Protocol* en AADL) associée au *thread controller_task.permanent*. A première vue, la modélisation AADL de la tâche de contrôle du pacemaker est triviale : le cœur a un fonctionnement périodique et il suffit d'utiliser une tâche périodique (*Dispatch_Protocol => periodic*). En faisant un peu plus attention, on s'aperçoit qu'il faut prendre en compte un certain nombre de propriétés temporelles supplémentaires par exemple : un battement dans l'oreillette doit être suivi d'une attente correspondant à l'intervalle oreillette-ventriculaire (AV). Pour éviter de consommer des ressources inutilement, une attente passive s'impose : la tâche libère la ressource de calcul (processeur) et sera réveillée lorsque le délai oreillette-ventriculaire sera écoulé. Cela signifie donc que le comportement de la tâche considérée n'est pas purement périodique puisque celle-ci s'endort et se réveille en cours d'activation. De plus, l'intervalle de réveil est programmable et variable (puisque'il peut dépendre des mesures de l'accéléromètre).

Quels types de tâche AADL pouvons nous utiliser ? La tâche que nous cherchons à modéliser se réveille sur évènements correspondant à une horloge interne, et la période globale de réveil n'est pas fixe. En conséquence, le rythme de réveil n'est pas constant et les tâches AADL *Periodic*, *Hybrid* et *Timed* sont à exclure. Une tâche sporadique pourrait convenir, mais nous n'avons pas ici de délai minimal constant entre deux réveils de la tâche. Nous optons donc pour une tâche apériodique, dont nous modélisons le comportement interne par le biais de l'annexe comportementale d'AADL.

Ensuite, nous ajoutons également des ports de données au *thread controller_task*. Ces ports vont permettre de configurer les paramètres temporels de la boucle de contrôle ; le nom de chaque port correspond au sigle du paramètre :

- la limite basse de stimulation (port de données LRL) ;
- la limite haute de stimulation (port de données URL) ;
- le délai d'attente atrio-ventriculaire (port de données AV) ;
- la période réfractaire de l'oreillette (port de données ARP), ou port ventriculaire du ventricule (port de données PVRP).

Notons également que la tâche *controller_task* possède un port *rate_value* (voir figure 12.3), dont les valeurs proviennent du traitement des informations fournies par l'accéléromètre.

Enfin, nous modélisons les modes de fonctionnement de la tâche de contrôle, ainsi que les changements de mode. Pour cela, nous ajoutons quatre modes au sein du *thread* AADL *controller_task.permanent* : DOOR, AAI, VDD, et VVT. Nous définissons ensuite quatre ports d'évènements (*to_DOOR*, *to_AAI*, *to_VDD*, *to_VVT*) et douze transitions de modes initiées par la réception d'un évènement sur l'un de ces ports : par exemple, *DOOR -[to_AAI] -> AAI* définit le changement de mode de *DOOR* vers *AAI* sur réception d'un évènement sur le port *to_AAI* de la tâche. Il est alors facile de déduire le contenu des onze autres changements de mode.

Pour modéliser le comportement de la tâche dans chaque mode, nous ajoutons un automate comportemental par mode de la tâche. Nous spécifions cela dans le modèle AADL en ajoutons la clause « *in modes* » à la fin de la définition de l'automate. Par exemple, la description AADL suivante correspond au comportement de la tâche de contrôle dans le mode AAI (listing ci-dessous).

```

thread implementation controller_task .permanent
  ...
  annex behavior_specification
  {**
  ...
  **} in modes AAI;
  ...
end controller_task .permanent;

```

Listing 12.1 – Annexe comportementale associée à un mode

Dans la suite de cette section, nous présentons le contenu des annexes comportementales qui correspondent au comportement de la tâche de contrôle dans chacun des sous-modes que nous avons choisi de modéliser. L'annexe comportementale permet de représenter le comportement de composants AADL sous la forme d'automates à états-transitions. Nous représenterons ces automates au travers de figures sur lesquelles les états seront représentés par des cercles et les transitions par des flèches. Les conditions et actions associées à chacune de ces transitions respecteront alors la syntaxe de l'annexe comportementale. En outre, bien que l'annexe comportementale permette de représenter différents types d'états (*initial*, *final*, *complete* et *execution*), nous n'utiliserons que deux types d'états :

- 1) un état *initial*, représenté par un petit cercle plein. Pour améliorer la lisibilité des modèles fournis, nous ignorons ici les phases d'initialisation du générateur de pulsations, ces étapes sont mise en œuvre lors du franchissement de la transition sortant de l'état initial de l'automate comportemental ;
- 2) des états *complete* représentent les différentes étapes du comportement de la tâche. Rappelons ici qu'un état *complete* correspond à un état dans lequel l'exécution de la tâche correspondante est suspendue. Comme nous allons le voir dans la suite de cette section, la fonctionnement de la tâche de contrôle du pacemaker consiste principalement à synchroniser l'arrivée et l'émission de différents types d'évènements

avec des exigences temporelles qui imposent un certain nombre d'attentes (avec mise en veille de la tâche).

Permanent/DOOR. L'automate comportemental représenté sur la figure 12.4 illustre le comportement de la tâche *controller_task.permanent* dans le sous-mode DOOR. Dans ce sous-mode, le comportement de la tâche de contrôle est relativement simple : après initialisation de la tâche, représentée sur la figure par la transition provenant du petit cercle plein, la tâche atteint l'état *Wait_rate_timeout* dans lequel son activité est suspendue. La tâche reste dans cet état jusqu'à être réveillée par l'échéance du délai d'attente imposé par la valeur sur le port de données *rate_value*. Le *thread* est alors réveillé, il envoie un événement sur le port *asp* (abréviation de *A_sense_and_pace* pour améliorer de lisibilité des figures) et atteint l'état *Wait_AV_Delay*. L'activité de la tâche est à nouveau suspendue lorsqu'elle atteint cet état, elle sera relancée après échéance du délai oreillette-ventriculaire (AV). La tâche envoie alors un événement sur le port *vsp* (abréviation de *V_sense_and_pace*) et atteint l'état *Wait_URL*, dans lequel elle attend qu'un délai suffisant soit écoulé pour respecter la contrainte *URL*. La tâche retourne alors dans l'état *Wait_rate_timeout*. Notons que la caractéristique LRL est assurée tant que *rate_value* représente des intervalles de réveil plus petits que ceux imposés par LRL.

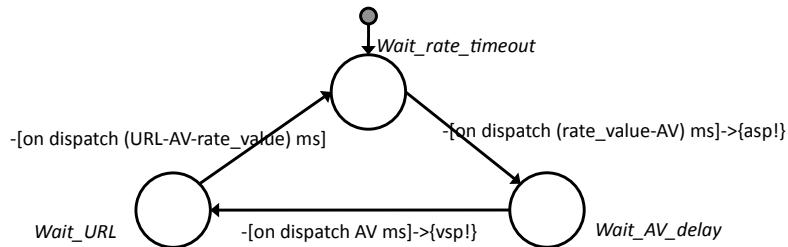


Figure 12.4 – Automate comportemental du générateur de pulsations dans le sous-mode DOOR

Permanent/AAI. La principale différence entre le mode *Permanent/AAI* et le mode *Permanent/DOOR* est le mécanisme d'inhibition et le respect de la période réfractaire de l'oreillette. Dans le mode AAI, l'inhibition est modélisée de la façon suivante : l'horloge interne de la tâche de génération de pulsations est réinitialisée lorsqu'un battement naturel est détecté dans l'oreillette. Le respect de la période réfractaire consiste, suite à un battement (naturel ou stimulé), à n'autoriser la détection d'un nouveau battement dans l'oreillette qu'après un délai fixe correspondant à la période réfractaire de l'oreillette (ARP).

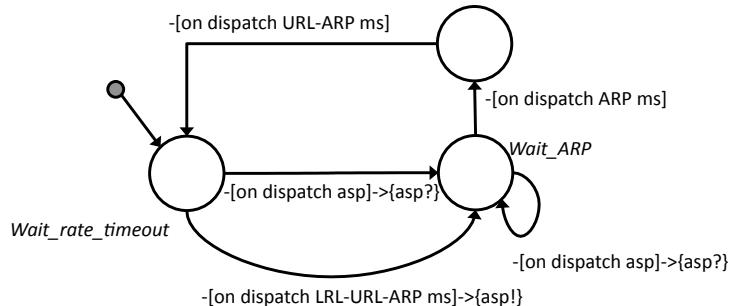


Figure 12.5 – Automate comportemental du générateur de pulsations dans le sous-mode AAI

La figure 12.5 représente le comportement décrit ci-dessus en utilisant les constructions fournies par l'annexe comportementale AADL. Lorsque le générateur de pulsations est activé dans le mode *AAI*, il atteint l'état *Wait_rate_timeout*, dans lequel la tâche de génération est inactive. Deux transitions permettent de sortir de cet état, d'activer la tâche, et d'atteindre un nouvel état de synchronisation appelé *Wait_ARP* :

- sur réception d'un évènement sur le port *asp* (pour *Atrial_sense_and_pace*), l'évènement reçu est alors consommé (*voir asp ?* sur la figure) ;
- sur échéance du délai *LRL-ARP*, une pulsation est alors émise sur le port *asp* (*voir asp !* sur la figure) qui correspond au port *Atrial_sense_and_pace*

Dans ce nouvel état d'inactivité, la tâche attend que le délai ARP soit écoulé : transition vers l'état *Wait_rate_timeout* avec pour condition *[on dispatch ARP ms]*. Si le générateur de pulsations, dans l'état *Wait_ARP*, reçoit un évènement de détection de battement dans l'oreillette, celui-ci est consommé et la tâche de génération reste dans l'état *Wait_ARP* : voir, sur la figure, la transition réentrant dans l'état d'attente du délai ARP.

Ce comportement implante bien le fonctionnement du pacemaker dans le mode *AAI* : le générateur de pulsations ne stimule le cœur que si aucune pulsation naturelle n'a été détectée avant l'échéance du délai de battement ; de plus, les battements détectés pendant la période réfractaire de l'oreillette sont ignorés.

La contrainte concernant le rythme minimal de battement est bien respectée par l'utilisation de la borne *LRL-ARP* : lorsque l'automate entre dans l'état *Wait_rate_timeout*, il s'est écoulé au plus *ARP ms* depuis le dernier battement dans l'oreillette. En stimulant le cœur après au plus *LRL-ARP ms*, on s'assure ainsi que les battements ont bien lieu au moins toutes les *LRL ms*.

Permanent/VVT. Dans le mode *Permanent/VVT*, dès qu'un battement est détecté dans le ventricule, le générateur de pulsations stimule ce même compartiment. Cependant, le générateur de pulsations doit respecter trois caractéristiques temporelles dans ce mode :

- le rythme minimal de battement (LRL), soit une période au bout de laquelle le compartiment ventriculaire sera stimulé ;
- la période réfractaire du ventricule (VRP), pendant laquelle les battements détectés sont ignorés ;
- le rythme maximal de battement (URL), ne considérant les mesures que si elles sont séparées par une période minimale.

On pourrait dans ce mode considérer l'opportunité de modéliser la tâche de contrôle par une tâche sporadique, puisque le rythme maximal de battement donne une borne supérieure sur l'intervalle de temps séparant deux réveils de cette tâche. Mais cette fois encore, cette caractéristique n'est pas une constante. Par conséquent, nous continuons de modéliser la tâche de contrôle par une tâche apériodique.

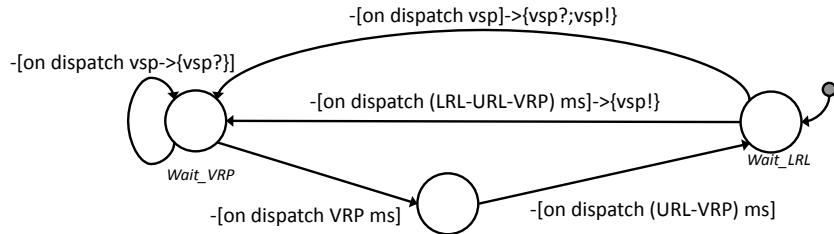


Figure 12.6 – Automate comportemental du générateur de pulsations dans le sous-mode VVT

La figure 12.6 représente le comportement de cette tâche de génération de pulsations dans le sous-mode VVT. De la même façon que pour les automates précédents, les états représentés dans cet automate sont des états de synchronisation : la tâche est inactive et attend d'être réveillée soit par la détection de battements dans le ventricule (transitions dont la condition est $-[on\ dispatch\ vsp]$), soit par l'échéance d'un des délais caractéristiques du pacemaker (transitions $-[on\ dispatch\ X\ ms]$ où X est égal à une opération mathématique sur les valeurs LRL, VRP et/ou URL).

L'automate comportemental du pacemaker dans le sous-mode VVT est légèrement plus complexe que ceux présentés précédemment, car le nombre de synchronisations à prendre en compte a augmenté. Il faut d'abord s'assurer du respect du rythme minimal de battement spécifier par la valeur LRL. Compte tenu des caractéristiques temporelles considérées dans ce mode, nous aurons consommé au plus URL+VRP ms au cours d'un cycle de battements. La condition pour quitter l'état d'attente du délai

maximal entre deux battements (*Wait_LRL*) est donc *-{on dispatch (LRL-URL-VRP) ms}*. Cette transition est inhibée si le pacemaker a entre temps détecté un battement naturel du cœur. Dans ce dernier cas, le ventricule est stimulé (ce qui correspond au comportement attendu : la détection d'un battement dans le ventricule entraîne la stimulation de ce dernier).

Une fois que l'une de ces deux transitions est franchie, l'automate attend l'échéance de la période réfractaire du ventricule, pendant laquelle les battements du cœur sont ignorés. La tâche attend ensuite que le délai correspondant au rythme maximal de battement (URL) soit atteint. A l'issue de ce délai, on retourne dans l'état *Wait_LRL*.

Permanent/VDD. Dans le mode *Permanent/VDD*, seul le ventricule peut être stimulé par le générateur de pulsations alors que les deux chambres sont sondées. Dans ce mode, le fonctionnement du pacemaker combine le comportement associé au mode VVI avec la contrainte temporelle AV. La mise en œuvre de ce mode nécessite de considérer quatre caractéristiques temporelles :

- le rythme minimal de battement (LRL), soit une période au bout de laquelle le compartiment ventriculaire sera stimulé ;
- la période réfractaire post-ventriculaire (PVRP), pendant laquelle les battements dans le ventricule sont ignorés ;
- le rythme maximal de battement (URL), ne considérant les mesures que si elles sont séparées par une période minimale ;
- le délai oreillette-ventricule (AV), période minimale séparant un battement dans l'oreillette d'un battement dans le ventricule.

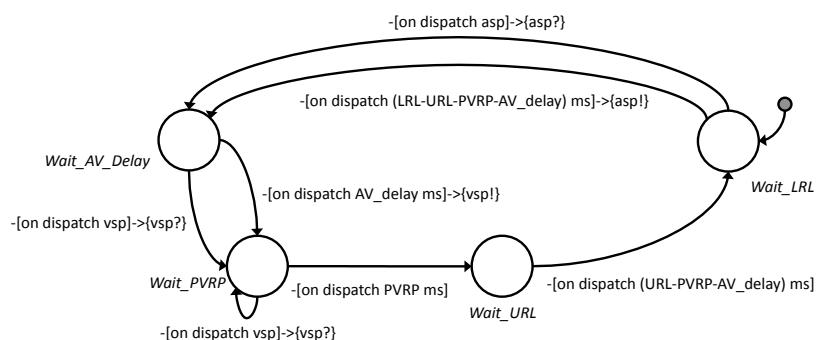


Figure 12.7 – Automate comportemental du générateur de pulsations dans le sous-mode VDD

La figure 12.7 représente le comportement de cette tâche de génération de pulsations dans le sous-mode VDD.

12.6. Modélisation du déploiement des fonctionnalités du pacemaker

Pour finaliser la modélisation AADL du pacemaker, il nous suffit de modéliser l'allocation des composants logiciels sur la plate-forme d'exécution : troisième et dernière étape de notre démarche de modélisation présentée en introduction de ce chapitre. Plus spécifiquement, nous devons associer les connexions logiques (entre tâches, entre processus, entre sous-programmes, etc.) aux connections physiques (bus de communication), les composants logiciels (processus, tâches, etc.) aux ressources de calcul (processeurs et mémoires associées) et les composants passifs (données) aux ressources de stockage (mémoire).

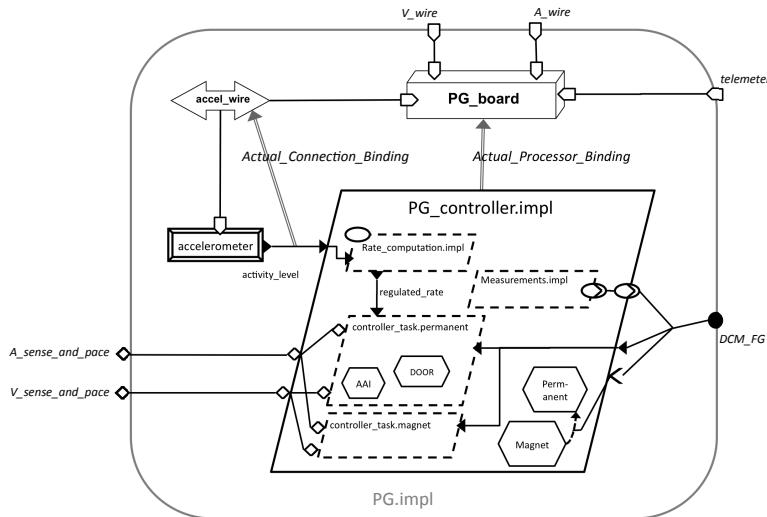


Figure 12.8 – Déploiement du processus de génération de pulsations

La figure 12.8 illustre le déploiement du processus de régulation des battements du cœur (*PG_Controller.impl*) sur la carte électronique *PG_board.impl* associée au sous-système *PG.impl*. Cette association, représentée par une double flèche, s'exprime en AADL textuel avec la propriété *Actual_Processor_Binding*. Par le biais de cette association, nous précisons alors que les calculs liés à la régulation des pulsations se feront sur la carte *PG_board.impl*. Cette association est représentée graphiquement par une double flèche.

De la même façon que précédemment, la figure 12.9 illustre l'association des connexions entre les différents sous-systèmes (générateur de pulsations, station de configuration et de supervision, et électrodes) aux médias de communication qui relient physiquement ces différents sous-systèmes (télémètres et fils électriques).

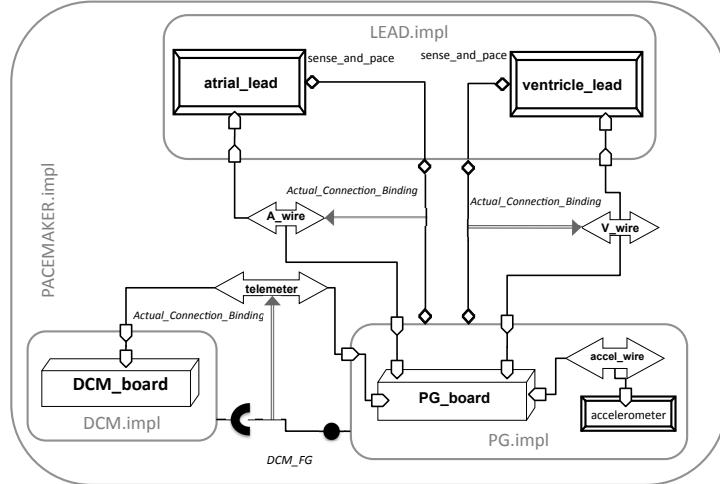


Figure 12.9 – Déploiement des connexions entre sous-systèmes du pacemaker

Nous avons ainsi obtenu une modélisation complète et relativement précise de l’architecture du pacemaker. Ce travail nous a demandé, en fonction des sous-parties de l’architecture, entre trois et cinq itérations. De nombreuses itérations seraient encore nécessaires pour obtenir une modélisation exhaustive du système. Il est important de noter ici que l’analyse de cette modélisation aide à mettre en œuvre ce processus car elle permet de détecter des fautes de conception avant même que la production du système n’ait commencé. L’exploitation des modèles décrits dans ce chapitre est présentée dans les chapitres 13 et 14.

12.7. Conclusion

Dans ce chapitre, nous avons proposé un modèle AADL du pacemaker, résultat de plusieurs itérations de modélisation et d’analyse du modèle. Ce chapitre illustre les difficultés liées à la modélisation d’un système, et propose quelques indications pour traiter ce type de problème. Les bénéfices de ce travail de modélisation seront présentés dans les chapitres suivants, qui s’intéressent à l’analyse et à la génération de code à partir de modèles AADL.

Le modèle que nous avons fourni ici est une spécification de l’architecture du pacemaker, incluant :

- la description du fonctionnement attendu ;
- le dimensionnement d’un pacemaker.

Les chapitres suivants vont montrer comment exploiter cette spécification pour :

- vérifier que l'architecture ainsi représentée respecte les exigences de réalisation d'un pacemaker (voir chapitre 13),
- produire une implémentation qui respecte cette spécification architecturale (voir chapitre 14).

Il est important de noter ici que les modèles proposés sont une spécification du pacemaker et n'imposent rien sur son implémentation, si ce n'est de respecter les contraintes imposées par le modèle (en termes de comportement et d'allocation par exemple). Pour illustrer ce propos, la présence de tâches impose que certains traitements soient réalisés tout en garantissant le respect d'échéances temporelles, mais n'impose pas que ces tâches soient implémentées par des « *threads systèmes* » d'un système d'exploitation.

12.8. Bibliographie

- [BAR 10] BAROLD S., STROOBANDT R., SINNAEVE A., *Cardiac Pacemakers and Resynchronization Step by Step : An Illustrated Guide*, John Wiley & Sons, 2010.
- [Bos 07] BOSTON SCIENTIFIC, PACEMAKER System Specification, janvier 2007.

Chapitre 13

Analyse à partir du modèle

13.1. Introduction

Dans ce chapitre, nous nous intéressons à l'analyse du modèle du pacemaker construit à l'aide d'AADL au chapitre précédent.

Cette phase d'analyse consiste à vérifier la cohérence du modèle de l'implémentation, et valider une partie des exigences système définies en amont de la conception. Les analyses pouvant être effectuées sont nombreuses, aussi nous nous concentrerons sur les exigences liées au respect des contraintes temporelles 22/P et 23/P.

Le langage AADL appartient à la famille des langages d'architecture possédant une sémantique riche correspondant à un domaine métier (l'ingénierie du logiciel embarqué). On parle alors de *Domain Specific Modelling Language* (DSML). Ainsi, l'implantation de l'analyse du modèle varie en fonction du degré d'indépendance entre l'implémentation de la méthode d'analyse et le langage. Nous distinguons trois cas :

- l'outil d'analyse a été développé spécifiquement pour analyser des modèles d'un langage dédié. Ainsi, le vérificateur de contraintes REAL [RUB 11] a été construit à seule fin d'analyser des propriétés statiques exprimées sur l'architecture du système ;
- la sémantique du modèle est extraite pour alimenter des outils spécifiques au domaine d'application (par exemple des outils d'analyse d'ordonnancement [SIN 05]). Ces outils permettent de valider efficacement un sous-ensemble clairement identifié des exigences du système, ici le respect d'échéances temporelles ;

Chapitre rédigé par Thomas ROBERT et Jérôme HUGUES.

- la sémantique du modèle est extraite afin d'alimenter des outils de vérification génériques (prouesseur logique, vérificateur CTL ou LTL d'automates ou réseaux de Petri).

Dans les deux derniers cas, la sémantique du DSML est extraite pour alimenter des outils de vérification. Toute tâche de vérification confronte un modèle à une exigence. Ainsi, il faut traduire les exigences du système dans le langage de contraintes de l'outil de vérification. Dans un second temps, il faut extraire du modèle AADL la fraction du modèle à analyser, formé d'un sous-ensemble des composants, connexions et propriétés du modèle pertinent pour l'analyse visée.

La suite du chapitre illustre les points suivants :

- validation des exigences (22)/P et (23)/P pour chacun des modes ;
- validation de l'exigence (23)/P lors d'un changement de mode.

Nous avons retenu l'outil UPPAAL, déjà présenté dans cet ouvrage comme support à notre démarche. Plutôt que de présenter un outil de transformation existant, nous avons opté pour la présentation de la démarche de construction du modèle, afin d'indiquer au lecteur les différentes étapes permettant de relier l'analyse d'un modèle AADL à la vérification d'un modèle UPPAAL.

13.2. Validation comportementale par mode et globale

Le but de cette section est de présenter en détail la phase d'extraction de la sémantique des automates comportementaux vers UPPAAL. La validation des exigences (22)/P et (23)/P sert à illustrer l'intérêt de la démarche, ses avantages et inconvénients.

Cette discussion permettra d'illustrer les problèmes liés à cette phase d'extraction par laquelle il faut passer dès qu'il existe un écart entre le formalisme de modélisation et de vérification. D'autres passerelles existent entre AADL et des formalismes de vérification comportementale [ABD 08, CHK 08, BER 09, BOZ 10]¹. Toutes ces passerelles suivent la même démarche, à des degrés divers de couverture du modèle d'entrée. Suivant les outils, la complexité du modèle peut être limité : monoprocesseur, réparti, prise en compte précise de la plate-forme matérielle. Il convient donc de connaître précisément les hypothèses de modélisation de l'outil, c'est-à-dire les modèles qu'il sait valablement traiter.

La transformation AADL vers UPPAAL couvrira naturellement les éléments qu'UPPAAL sait valablement traiter, et aux propriétés qui nous intéressent. Il exclura les propriétés et éléments qu'UPPAAL ne sait exprimer. Ici, nous nous intéressons plus

1. Une liste plus complète d'outils est disponible sur le site du comité AADL : www.aadl.info

particulièrement au comportement des tâches du système, leur interaction et leur comportement temporel. Nous nous restreignons donc naturellement aux seuls composants du modèle ayant un impact sur ces propriétés comportementales.

13.2.1. Contexte de validation et raffinement des exigences

Les modes, les automates comportementaux et les tâches possèdent une sémantique similaire à celle d'un réseau d'automates communicants. Ce constat est lié à la sémantique des connexions entre composants, aux ports d'entrée/sorties et à l'automate comportemental standard d'un *thread* AADL. Cependant, un tel formalisme ne permet pas de tirer facilement partie de connaissances telles que les latences de propagation de messages, et l'accès à une base de temps commune. De plus, l'analyse de tels modèles pose de nombreux problèmes de décidabilité [BRA 83] – c'est-à-dire si l'on peut construire un algorithme permettant de fournir une réponse à la question posée par un enchaînement d'actions élémentaires.

Il est donc priomordial d'utiliser des modèles de concurrences pour lesquels l'analyse d'accessibilité d'un état soit décidable. Cette propriété est nécessaire dès le moment où l'on cherche à vérifier des propriétés de « safety » par l'analyse de modèles [ALP 87, KUP 99].

La traduction vers UPPAAL a été choisie pour la simplicité de son modèle de réseaux d'automates et sa capacité à modéliser des transitions conditionnelles contrignant l'écoulement du temps (actives uniquement dans certaines conditions). Enfin, le modèle est décidable et complètement outillé. Un tutoriel complet de mise en pratique de la vérification formelle sous UPPAAL est disponible sur le site de l'outil [BEH 04].

13.2.2. Traduction des automates comportementaux vers UPPAAL

UPPAAL permet de vérifier des propriétés exprimées en logique temporelle (CTL) sur un réseau d'automates dits temporisés [ALU 99] étendus. L'espace d'états des formules (CTL) est formé de variables booléennes et des variables entières bornées.

Les automates temporisés. Un automate temporisé est constitué d'états discrets appelés lieux et de transitions conditionnelles reliant ces états :

l'état d'un automate est caractérisé par le lieu qu'il occupe ; et un ensemble de variables « réelles » correspondant à des horloges. Ces variables croissent librement pour représenter l'écoulement du temps et peuvent être réinitialisées pour mesurer l'écoulement du temps depuis un événement donné. Ces horloges sont utilisées pour définir les conditions de franchissement des transitions. UPPAAL

étend ce modèle par des variables entières bornées. Ces dernières varient lors d'un franchissement de transition. La modification s'exprime à partir d'un opérateur d'affectation et d'un langage simple d'expressions arithmétiques ;

les transitions sont franchissables à la condition que l'état de l'automate (lieu occupé, horloges et variables entières) valide une condition logique simple. Cette condition est appelée garde. La garde d'une transition peut être triviale (toujours vraie) ou construite sous la forme d'une conjonction de contraintes élémentaires (par exemple $variable < Constante$) ;

UPPAAL permet de contraindre la manière dont le temps s'écoule dans un « lieu ».

Cela peut aller de : « le temps ne peut progresser » pour représenter des situations d'enchaînement instantanées (lieux dit *Urgent*), jusqu'à « l'horloge x doit rester inférieure à $Constante$ » pour exprimer le fait qu'une action doit nécessairement être exécutée avant d'atteindre cette valeur.

La figure 13.1 illustre le cas d'un automate représentant le comportement de la tâche *controller_task* dans le mode DOOR.

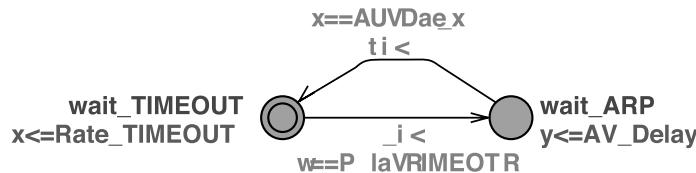


Figure 13.1 – Automate temporisé équivalent à l'automate AADL pour DOOR 12.4

Ce modèle définit deux lieux : *wait_TIMEOUT* et *wait_ARP*. Chaque lieu dispose d'une horloge x et y . L'automate se trouve dans un lieu tant qu'une condition est vérifiée $x < Rate_TIMEOUT$, et change d'état lorsque l'horloge atteint l'instant *Rate_TIMEOUT*. Le signal *ap* est alors émis.

Définition du problème de traduction. Par la suite, nous souhaitons fournir une solution au problème de traduction de modèles AADL vers des automates UPPAAL. Dit autrement, nous cherchons à planter le schéma décrit dans la figure 13.2.

La figure illustre le processus de traduction des modèles AADL et des exigences systèmes vers UPPAAL de façon à réaliser une vérification « équivalente » sous UPPAAL à celle qui aurait pu être menée directement sur les automates comportementaux du langage AADL. La démarche n'a d'intérêt que si une propriété fausse dans AADL l'est nécessairement sous UPPAAL. On parle alors de correction de la démarche. Inversement, si une propriété est valide dans AADL, elle ne doit pas engendrer de verdict contradictoire sous UPPAAL. Ce second aspect permet de définir la précision de la démarche.

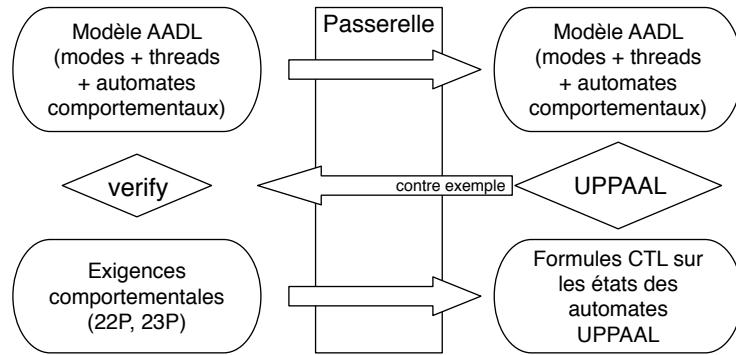


Figure 13.2 – Traduction de modèles entre AADL et UPPAAL.

Ansi, si l'on souhaite pouvoir définir la transformation vers UPPAAL, nous devons résoudre deux problèmes :

- définir la sémantique des automates comportementaux d'AADL sous forme d'automates états/transitions ;
- « traduire » les scénarios que l'on souhaite évaluer sous forme de prédictats CTL, utilisés par UPPAAL pour exprimer les exigences à vérifier.

Un point important est de garantir la traçabilité entre modèles AADL et UPPAAL pour assurer l'équivalent entre modèles. Cela se fait le plus souvent par des conventions de nommage des entités générées.

Sémantique des automates comportementaux. La validation des exigences 22/P et 23/P passe par l'analyse des éléments du modèle qui définissent le comportement de la tâche *controller_task* sur le long terme. Le comportement d'une tâche dépend des éléments suivants :

- modèle d'ordonnancement/concurrence des tâches et de l'état de l'ordonnateur ;
- modèle de communication intertâches et l'état des tampons de communication ;
- synchronisation entre ports de communication et tâches ;
- synchronisation des horloges et comportement des tâches ;
- impact des modes et des transitions de mode.

La question devient : « Comment intègre-t-on toutes ces informations dans un système UPPAAL ? ». Doit-on créer une description tenant compte de tous ces éléments ? Nous passons en revue ces différents problèmes et indiquons des éléments de réponse.

L'ordonnancement – Nous allons dans un premier temps évacuer la question de la modélisation de l'ordonnancement des tâches. Ces tâches sont caractérisées par des attributs définissant leur périodes, des relations de priorité et de causalité. Nous nous intéressons à la validation de la cohérence de ces paramètres vis-à-vis des exigences de haut niveau. La faisabilité de l'ordonnancement et du déploiement est hors spectre de la traduction vers UPPAAL.

Pour prendre en compte l'ordonnancement, il suffit de modéliser les intervalles où la tâche peut émettre des événements de stimulation. Ceci permet de prouver que le comportement de la tâche vérifie les exigences de haut niveau.

Les ports et le modèle de synchronisation – Les communications AADL ont un impact sur le comportement d'un assemblage de composants actifs (tâches). Il existe différents degrés de synchronisation entre les opérations de « lecture/écriture » sur un port et les événements effectifs de réception et envoi, ainsi que les comportements associés à la réception de ces données. En particulier, ces deux événements peuvent être largement désynchronisés. Il convient donc de modéliser ce mécanisme de synchronisation potentiellement complexe.

Trois éléments doivent être modélisés pour représenter un port de communication : la file d'attente (par défaut de taille 1), le protocole de gestion des dépassemens de la file et la synchronisation avec les tâches. Nous faisons les choix suivants :

- le protocole de gestion des dépassemens de type *dropNewest* sera considéré. En cas de dépassement de la capacité de la file, les nouveaux messages seront considérés comme perdus et un événement d'erreur sera généré ;
- la synchronisation avec les tâches correspond au déclenchement d'une activité (ce qui introduit une relation de causalité entre les deux événements) ;
- la modélisation des ports se fera à travers la définition d'un automate servant à représenter le fonctionnement du port de communication. Cet automate sera synchronisé avec l'automate de la tâche à laquelle il appartient. Une synchronisation supplémentaire sera nécessaire pour représenter la relation de causalité correspondant à l'envoi et la réception d'un message.

De ces choix, nous en déduisons les patrons de modélisation UPPAAL :

- la prise en compte de la file d'attente se fait en représentant chaque état de remplissage de la file d'attente. Dans le cas standard, il y a deux états correspondants aux conditions suivantes : « file vide » et un « événement présent » ;
- la prise en compte du protocole de gestion de dépassement du tampon se fait en rajoutant des transitions permettant de représenter le cas où un événement entraîne la production d'un autre événement (par exemple une erreur) ;

– la prise en compte de l'accès au contenu du port se fait en rajoutant un événement spécifique permettant de synchroniser le port et la tâche. Nous noterons cet événement par le nom du port suivi de « _in » pour les ports d'entrée.

Dans le cas de la tâche *controller_task*, nous obtenons ainsi quatre automates. L'automate de la figure 13.3 représente l'un des automates de port d'événements.

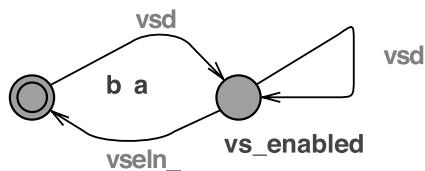


Figure 13.3 – Modèle d'un port et ses événements de synchronisation

Nous devons à présent traiter le cas des valeurs échangées par des ports de données. Nous rencontrons là le premier obstacle dans l'usage de UPPAAL pour réaliser la vérification du système. Ces ports de données sont utilisés soit pour représenter des constantes de configuration, soit des valeurs variant dans le temps. Nous allons nous focaliser sur le cas où seules des constantes de configuration sont utilisées, par exemple dans les modes VVT et VOO. Le tableau page 28 de la spécification du pacemaker permet de comprendre quels paramètres jouent effectivement un rôle dans chaque mode. Dans ce cas, nous représenterons ces valeurs comme des constantes passées en paramètres au moment de la définition des automates temporisés d'UPPAAL. Ceci va nous permettre de modéliser assez simplement un dispatch sur « timeout » dans le cas d'une tâche hybride (cas non couvert par de nombreuses passerelles existantes). Nous faisons ce choix de modélisation car ces paramètres sont utilisés dans des gardes impliquant des horloges et ne peuvent changer au cours du temps.

Le tableau précisant les intervalles de variations autorisés pour chaque paramètres permet de mettre en place des scripts définissant les différentes valeurs pour TIMEOUT et SAFETY_DELAY, ainsi que VRP. Ces scripts permettent d'engendrer un automate pour chaque configuration de paramètres.

Les conditions d'activation – Cette étape de transformation vers UPPAAL repose sur l'interprétation des états des automates comportementaux d'AADL dit « complete state » et des transitions avec « dispatch condition ».

On distingue deux conditions d'activation (ou de *dispatch*) :

- une condition sur expiration d'une échéance relative à l'entrée dans l'état final ;
- une condition sur la présence d'un élément dans la file d'un port d'événements.

Dans le premier cas, nous appliquerons la transformation décrite par la figure 13.4. La projection proposée repose sur une traduction de différents éléments unitaires :

- l'émission d'un événement *action* au niveau AADL sera traduit en un événement UPPAAL *action_out* ;
- la condition d'activation sera traduite en UPPAAL en : a) un invariant forçant le tirage d'une transition à l'expiration de l'échéance, $X == Y$; b) une garde empêchant la transition de « *dispatch* » temporisé de la tâche d'être tirée strictement avant son échéance, $X <= Y$; c) et la mise à zéro d'une horloge pour mesurer le temps écoulé depuis l'entrée dans le *complete state* représentant l'attente avant déclenchement de l'activité, liens $X = 0$.

Il est à noter qu'une seule horloge est nécessaire pour traiter l'ensemble des transitions de ce type pour chaque automate. C'est un point important : un grand nombre d'horloges rend plus complexe l'analyse d'accèsibilité par UPPAAL.

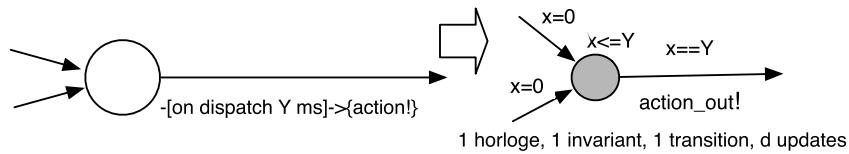


Figure 13.4 – Transformation d'une transition de *dispatch* sur « *timeout* »

Dans le second cas, correspondant à des sources de *dispatch* reliées aux ports d'événements, nous utilisons des transitions synchronisées avec les automates représentant les ports *in* ou *in out*. La figure 13.5 représente l'automate de la tâche de contrôle et ses automates de ports. Il est à noter la combinaison de l'usage de l'expansion d'un *dispatch* déclenché par le temps (zone 1), avec les *dispatches* déclenchés sur événement (zone 2).

13.2.3. Raffinement des exigences 22-23/P

La première étape lors d'une tâche de validation est de lever au maximum l'ambiguïté contenue dans les exigences définissant le comportement attendu du système. Les exigences 22 et 23 utilisent différents concepts sans nécessairement les définir et introduisent une certaine liberté d'interprétation. Tout d'abord, nous rappelons les définitions :

contrainte LRL (rappel) : dans les modes DXX et VXX, la limite basse de stimulation (ou LRL – *Lower Rate Limit*) démarre lorsqu'une pulsation du ventricule est détectée, ou qu'une stimulation est détectée et correspond toujours à un taux minimal de pulsations par minute ;

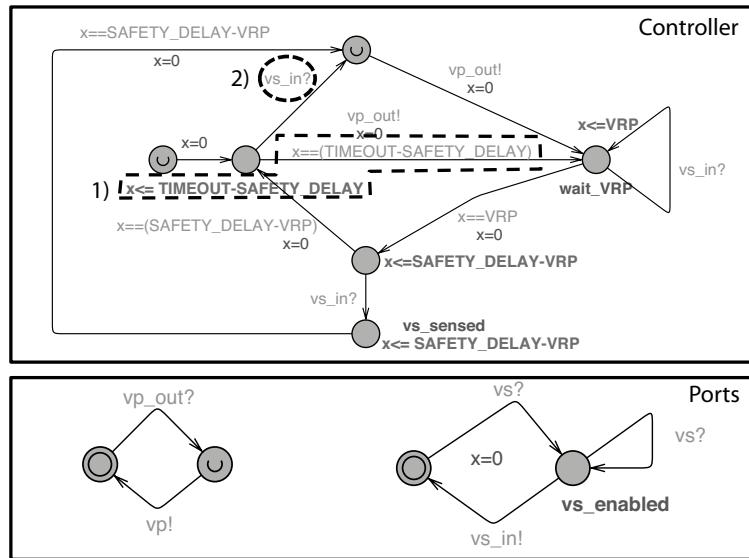


Figure 13.5 – Modèle UPPAAL d'une tâche et de ses ports (mode VVT)

contrainte URL (rappel) : la limite haute de pulsation (URL – *Upper Rate Limit*) est le nombre maximum d'événements mesurés au niveau de l'oreillette. De manière duale, l'URL est l'intervalle minimum entre un événement du ventricule et la prochaine stimulation du ventricule.

Les contraintes sur les taux de battements peuvent être interprétées en termes de délais minimaux/maximaux entre deux événements. Cette approximation est une interprétation correcte d'un point de vue sûreté. Il s'agit d'une réinterprétation d'une propriété mathématique.

Par la suite, nous utilisons TIMEOUT pour désigner le délai maximal autorisé. Cette quantité nous permet d'exprimer la condition suffisante de l'exigence 22/P, et SAFETY_DELAY la durée minimale entre deux événements de stimulation sur les sondes du pacemaker, permettant d'exprimer la condition suffisante à 23/P.

Ceci nous permet de raffiner ces deux exigences (nous ajoutons le suffixe « /R » pour identifier le raffinement et tracer l'héritage existant entre 22/P et 22/P/R) :

- 22/P/R : La durée entre deux événements de stimulation (naturels et détectés ou artificiels et engendrés) doit être inférieure à TIMEOUT ;
- 23/P/R : la durée minimale entre un événement de stimulation (artificielle) et l'événement de stimulation artificielle ou naturelle précédent doit être supérieure à SAFETY_DELAY pour un même élément du cœur (oreillette ou ventricule).

13.2.4. Etude du modèle permanent/VVT

En appliquant les principes évoqués ci-dessus, nous avons obtenu l'automate du mode permanent/VVT présenté dans la figure 13.5.

Nous devons à présent exprimer les contraintes servant à vérifier les exigences 22/P/R et 23/P/R dans CTL. Nous avons choisi la même méthode que celle proposée pour l'analyse avec MARTE : les observateurs formels.

Les observateurs résultants de ces exigences sont assez simples (figure 13.6). Ces exigences concernent la réception et l'émission d'événements sur les électrodes du système. Pour 22/P, nous rebouclons sur un même état, et remettons l'horloge à zéro sur réception de l'événement vs ou vp . Si aucun événement n'est détecté au bout de TIMEOUT, alors nous allons dans un état d'erreur. L'automate pour 23/P suit la même logique, seules les gardes changent : une erreur est détectée si vp est reçu avant le délai minimal SAFETY_DELAY. L'horloge y qui mesure l'arrivée des événements vp est remise à zéro dès lors que les événements sont suffisamment espacés dans le temps.

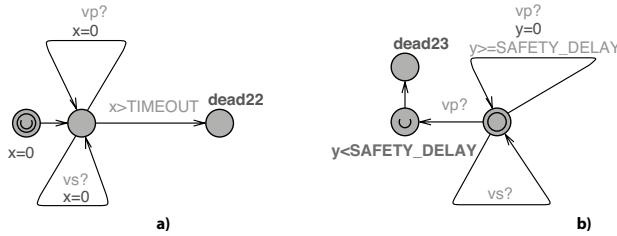


Figure 13.6 – Observateurs : a) exigence 22/P pour VVT, b) exigence 23/P

Complexité et explosion combinatoire. Pour réaliser la vérification sur l'ensemble des configurations acceptables du mode VVT un script affecte les valeurs des constantes TIMEOUT, VRP et SAFETY_DELAY puis démarre la vérification. Nous procédons à une discréttisation du temps pour tester différentes configurations. La vérification de chaque configuration prend entre cinq et dix secondes. Nous tombons néanmoins sur un problème d'explosions combinatoires du fait du nombre de configurations possibles, de l'ordre de 10 000. Ce chiffre est très élevé, à comparer à la relative simplicité des modèles considérés.

13.2.5. Etude du changement de mode permanent/VVT→Magnet/VOO

Le modèle AADL de la section 12.4.1 définit les changements de modes de fonctionnement autorisés pour le générateur d'impulsions. Les modes permettent entre autres

de définir quelles structures actives s'exécutent effectivement sur le matériel à un instant donné. C'est cette abstraction qui a été utilisée pour séparer les différents modèles comportementaux déduits des modes opérationnels du PaceMaker (DXX, VXX, etc.).

Changement de mode et exigences. Le modèle considéré jusqu'à présent représentait le comportement de la tâche de génération d'impulsions dans le mode de fonctionnement VVT. Nous nous concentrerons sur le problème de changement de mode associé à la tâche de contrôle de la génération d'impulsions. Chaque changement de mode possède un mode de destination auquel est associé un modèle de tâche différent (au moins en ce qui concerne la clause de description du comportement).

Par défaut, une tâche ne peut changer de modes qu'à partir du moment où elle est dans un état de type *complete state*. Chaque changement de mode est associé avec un port d'événements permettant de déclencher la transition. Une tâche nouvellement activée suite à un changement de mode possède toujours la même procédure d'activation. Si un état doit être transféré lors du changement de mode, il faut le gérer avant d'envoyer l'événement sur le port de changement de mode.

Lors du passage du mode Permanent au mode magnet (au niveau du processus représentant le pacemaker), la tâche de contrôle doit passer du mode VVT au mode VOO. Nous allons vérifier si la contrainte LRL reste valide malgré le changement de mode.

Les automates comportementaux proposés dans le modèle contiennent uniquement des états de type *complete state*. Par défaut, le changement de mode est donc autorisé en chacun de ces états. Un ensemble de transitions est ajouté au modèle existant pour représenter le changement de mode : une par état équivalent à un *complete state*. Ces transitions permettent de suspendre l'activité du modèle associé au mode VVT suite au changement de mode modélisé par la réception de l'événement *to_VOO*. Par défaut le changement de mode peut avoir lieu depuis un *complete state* uniquement lorsque le port d'événements associé contient un élément. Lors de l'activation du mode de destination, le modèle de tâche associé démarre son activité depuis l'état initial.

Vérification et bilan. Nous avons réalisé les deux modèles correspondant et démontré la violation possible de l'exigence (22)/P et (23)/P lors du processus de changement de mode. Ce défaut provient de l'absence d'un mécanisme, dans la version courante du modèle, permettant de conserver la mémoire de la dernière date de *dispatch* à travers le changement de mode. Or cette connaissance est nécessaire pour contrôler les transitions dans le mode VOO (pour le respect de 22/P aussi bien que de 23/P). Les contre-exemple d'exécution nous placent dans la situation où le changement de mode est effectué à une date différente d'un *dispatch*. Les deux scénarios suivants sont possible :

- le changement de mode a lieu juste après une stimulation du ventricule ;

- le changement de mode a lieu juste avant l’expiration du délai TIMEOUT (associé à la contrainte sur LRL).

La tâche nouvellement activée suite au changement de mode ne peut savoir combien de temps s'est écoulé depuis le dernier instant d'activation. Par défaut, il est nécessaire de déterminer quand la nouvelle impulsion doit avoir lieu. Cependant, aucune valeur fixe ne peut assurer à la fois 22/P et 23/P en même temps dans tous les cas de figure. L'usage d'UPPAAL nous a permis de visualiser ce problème dû à la variabilité du moment où la tâche peut changer de mode.

13.3. Conclusion

Dans ce chapitre, nous nous sommes intéressés à la vérification des exigences 22/P et 23/P du pacemaker, en prenant comme entrées les modèles AADL construits précédemment.

La vérification et la validation d'un système aussi complexe que le Pacemaker nécessite la définition d'une méthodologie précise. Nous avons pris le parti de présenter les étapes permettant de traduire un modèle AADL en un modèle UPPAAL. Cet exemple vise avant tout à montrer les questions à se poser lors d'une démarche IDM visant à traduire un modèle en une spécification formelle en vue de son analyse.

La démarche repose sur un triple questionnement :

- quelle est la propriété à analyser ? De la nature de la propriété découlera le choix d'une technique d'analyse, formelle ou non ;
- quel est l'outil permettant de supporter cette analyse ? L'exemple du pacemaker montre une très grande variabilité de paramètres et de très nombreuses configurations à traiter. Cela vient des limites intrinsèques à l'outil, mais surtout des techniques actuelles : le problème général étant indécidable, nous l'avons ramené à une famille de problèmes paramétriques qui unitairement le sont ;
- comment traduire un modèle en une spécification formelle ? Nous avons montré dans un premier temps qu'il est nécessaire d'isoler les éléments du modèle d'entrée pertinents pour l'analyse et l'outil. Partant de cette liste, nous avons fourni des éléments de traduction.

Ce faisant, nous avons posé les bases d'un traducteur AADL vers UPPAAL, que nous avons ensuite appliqué à notre problème de vérification.

Du fait de la nature du problème, reposant sur des intervalles de temps, nous avons opté pour une discréttisation des paramètres permettant d'analyser différentes configuration. Nous avons d'une part montré que le système, pour un mode donné, vérifiait

ses exigences ; d'autre part nous avons mis en évidence des situations où les exigences n'étaient pas remplis.

Dans ce dernier cas, il faut alors s'interroger : qui du modèle, de l'exigence, de la traduction ou de l'outil est erroné ? Nous avons montré qu'il s'agissait d'un défaut structurel : une information (date de la dernière activation) est perdue lors du changement de mode. Il s'agit ici d'un élément spécifique au langage AADL et sa définition d'un changement de mode. Pour autant, cette définition est compatible avec bon nombre de pratiques industrielles, il convient aussi aux exigences définissant le système de clarifier la notion de mode, et d'indiquer précisément les éléments préservés ou modifiés lors d'un changement de mode, et les propriétés que l'on accepte de violer lors de ce changement de mode.

Nous laissons au lecteur le soin de porter une réponse à cette question, source de nombreux débats, que ce soit au sein de communautés des langages de spécification, des ingénieurs système expert du domaine technique et des implantateurs d'outils.

Il convient de toujours se souvenir que la vérification d'un modèle permet de supprimer les fautes de conception du modèle, et rien d'autres !

13.4. Bibliographie

- [ABD 08] ABDOUL T., CHAMPEAU J., DHAUSSY P., PILLAIN P. Y., ROGER J.-C., « AADL Execution Semantics Transformation for Formal Verification », *International Conference on Engineering of Complex Computer Systems (ICECCS)*, IEEE Computer Society, p. 263-268, 2008.
- [ALP 87] ALPERN B., SCHNEIDER F. B., « Recognizing safety and liveness », *Distributed Computing*, vol. 2, p. 117-126, 1987.
- [ALU 99] ALUR R., « Timed Automata », *Lecture Notes in Computer Science*, vol. 1633, p. 8–22, 1999.
- [BEH 04] BEHRMANN G., DAVID A., LARSEN K. G., « A Tutorial on Uppaal », BERNARDO M., CORRADINI F., Eds., *Formal Methods for the Design of Real-Time Systems, International School on Formal Methods for the Design of Computer, Communication and Software Systems, SFM-RT 2004, Revised Lectures*, vol. 3185 de *Lecture Notes in Computer Science*, Springer, p. 200-236, 2004.
- [BER 09] BERTHOMIEU B., BODEVEIX J.-P., CHAUDET C., ZILIO S. D., FILALI M., VER-NADAT F., « Formal Verification of AADL Specifications in the Topcased Environment », *International Conference on Reliable Software Technologies - Ada-Europ*, n° 5570Incs, Brest, France, Springer-Verlag, p. 207-221, 2009.
- [BOZ 10] BOZZANO M., CIMATTI A., KATOEN J.-P., NGUYEN V., NOLL T., ROVERI M., WIMMER R., « A Model Checker for AADL », TOUILI T., COOK B., JACKSON P., Eds., *Computer Aided Verification*, vol. 6174 de *Lecture Notes in Computer Science*, p. 562-565, Springer, Berlin , Heidelberg, 2010.

- [BRA 83] BRAND D., ZAFIROPULO P., « On Communicating Finite-State Machines », *Journal of the ACM*, vol. 30, n° 2, p. 323–42, avril 1983.
- [CHK 08] CHKOURI M. Y., ROBERT A., BOZGA M., SIFAKIS J., « Translating AADL into BIP - Application to the Verification of Real-Time Systems », *MoDELS Workshops*, p. 5-19, 2008.
- [KUP 99] KUPFERMAN O., VARDI M. Y., « Model Checking of Safety Properties », HALBWACHS N., PELED D., Eds., *Computer Aided Verification, 11th International Conference, (CAV)*, vol. 1633 de *Lecture Notes in Computer Science*, Springer, p. 172-183, 1999.
- [RUB 11] RUBINI S., SINGHOFF F., HUGUES J., « Modeling and Verification of Memory Architectures with AADL and REAL », *Engineering of Complex Computer Systems (ICECCS), 2011 16th IEEE International Conference on*, p. 338-343, avril 2011.
- [SIN 05] SINGHOFF F., LEGRAND J., NANA L., MARCÉ L., « Scheduling and memory requirements analysis with AADL », *Ada Lett.*, vol. XXV, p. 1-10, ACM, novembre 2005.

Chapitre 14

Génération de code à partir du modèle

14.1. Introduction

Ce chapitre décrit le processus de production d'applications temps réel embarquées dont les étapes sont présentées en figure 14.1. Le point de départ est un modèle en AADL décrivant une application TR²E. La majeure partie de ce modèle est écrite par l'utilisateur en respectant des restrictions à ce langage définies dans [ZAL 08]. Il s'agit d'un sous-ensemble du langage AADL qui ne comporte pas les constructions non déterministes ou de complexité trop importantes pour les systèmes TR²E. Ce sous-ensemble a été défini en s'inspirant du profil Ravenscar pour le développement des systèmes critiques [BUR 04].

Pour spécifier les propriétés non architecturales des composants décrits, le développeur utilise les propriétés standards du langage AADL. En cas d'absence de propriétés standards spécifiant une caractéristique particulière (priorité d'une tâche, catégorie d'un type de donnée, etc.), le développeur utilise un ensemble de propriétés qui complète les ensembles standards (définies également dans [ZAL 08]).

Ces dernières propriétés ainsi que d'autres manquantes (priorité, etc.) ont été intégrées dans la toute récente version du standard, AADLv2. Toutes ces propriétés enrichissent l'expressivité du modèle et permettent de spécifier de nombreuses caractéristiques des composants :

- caractéristiques fonctionnelles comme les noms des implantations des sous-programmes AADL donnés sous forme de fichiers sources ou de bibliothèques ;

Chapitre rédigé par Laurent PAUTET et Bechir ZALILA.

- caractéristiques non fonctionnelles comme les périodes des *threads* ou leurs échéances ;
- caractéristiques architecturales comme l’association entre les processus et les processeurs sur lesquels ils seront exécutés.

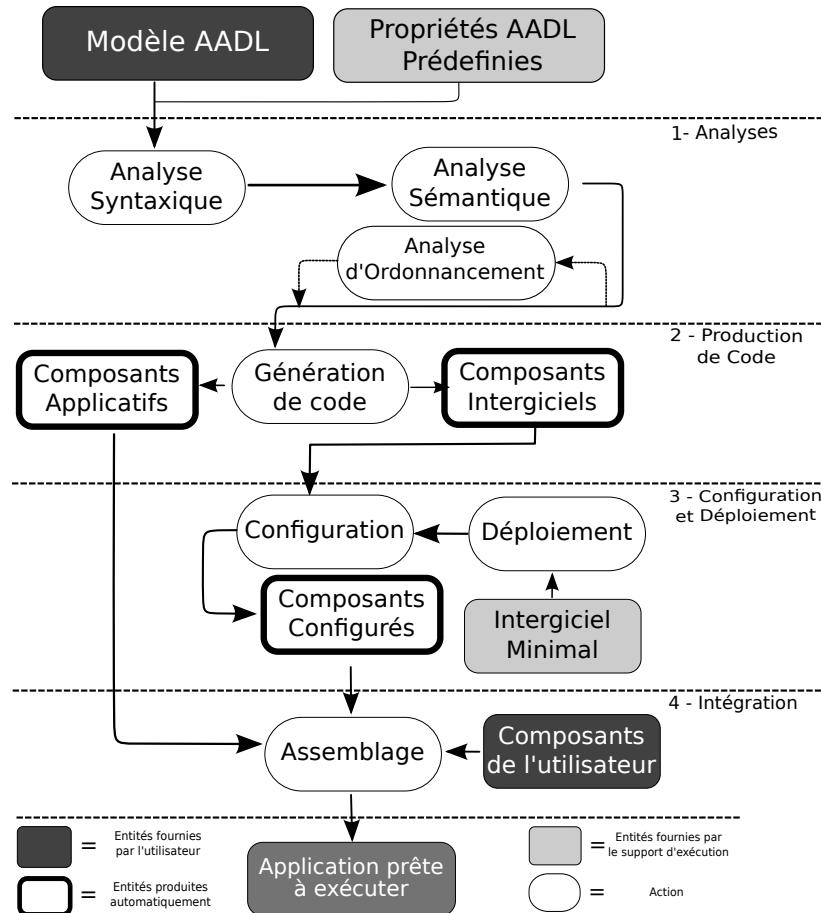


Figure 14.1 – Processus de production détaillé

Comme le montre la figure 14.1, le processus de production que nous proposons est constitué de quatre étapes principales.

Analyses. Une fois l’application TR²E modélisée en AADL, son modèle subit une suite d’analyses afin de garantir que le modèle est cohérent. Ces analyses consistent

à vérifier la syntaxe et la sémantique du modèle vis-à-vis du standard AADL. Ces étapes ont été décrites dans les chapitres 11 et 12. Ensuite, les analyses décrites dans le chapitre 13 sont appliquées sur les modèles.

Nous vérifions aussi que le modèle spécifié se limite au sous-ensemble du langage AADL mentionné plus haut. Ceci permet par la suite d'avoir un code généré conforme au profil Ravenscar [BUR 04]. Des analyses optionnelles effectuées par des outils externes sont également possibles (comme l'analyse d'ordonnancabilité). Ces analyses effectuées sur le modèle AADL renforcent la fiabilité de l'application.

Production de code. Après la phase d'analyse, le modèle AADL est utilisé pour produire automatiquement deux familles de composants de l'application temps réel ré-partie embarquée :

- les composants applicatifs qui rassemblent généralement les types de données ainsi que les sous-programmes utilisés par l'utilisateur. Ces composants sont produits automatiquement à partir du modèle AADL, notamment à partir des composants **data** et **subprogram**. Les produire automatiquement, avec des conventions de nommage précises et connues par l'utilisateur a un double bénéfice. Tout d'abord, il permet d'avoir une traçabilité du code tout au long du processus de développement. En effet, il est facile de retrouver les entités de code source qui ont été générées à partir d'une entité AADL : elles ont généralement le même nom ou des noms très proches. Cette traçabilité s'avère nécessaire dans le cadre de processus de certification de certaines familles d'applications embarquées critiques. Ensuite, ceci permet d'utiliser les entités générées automatiquement dans d'autres endroits du code généré mais aussi dans le code écrit par l'utilisateur. Les règles de transformation que nous utilisons sont similaires aux règles de projection des descriptions IDL de CORBA vers un langage de programmation spécifique ;

[Corriger toutes ces fontes ?](#)

- les composants intergiciels fortement personnalisables en fonction des propriétés de l'application. Ces composants sont produits automatiquement en fonction de la topologie de l'application et en particulier l'interaction entre les composants de types **process** et **thread**. Leur optimisation en fonction de la plate-forme est effectuée en analysant les composants matériels présents dans le modèle (**processor**, **bus**, etc.).

Configuration et déploiement. Cette étape consiste à sélectionner les services de l'intergiciel (dit autrement la plate-forme d'exécution AADL) qui sont utilisés par un noeud donné. La sélection est effectuée automatiquement grâce aux informations de déploiement extraites en particulier des composants matériels du modèle AADL (caractéristiques des composants **processor** et **bus** par exemple). Ces services sélectionnés, ainsi que ceux qui générés automatiquement, sont configurés en fonction des propriétés du noeud. A l'issue de ces deux étapes est produit un ensemble de services intergiciels optimisés et configurés selon les besoins de chacun des noeuds de l'application. Un service est un élément architectural du noyau de l'intergiciel réalisant

une étape de la communication entre deux nœuds en appelant les routines correspondants du système d'exploitation. On trouve par exemple le service de protocole qui permet de gérer une communication selon un protocole donné (IIOP, etc.), le service d'exécution qui permet d'exécuter sur le serveur les activités voulues par le clients, etc. Généralement, les services peuvent être paramétrés selon les besoins d'une application. Certains services peuvent même ne pas être utilisés. Pour cela, ils sont isolés pour pouvoir les produire et les configurer indépendamment les uns des autres.

Intégration. Cette phase consiste à rassembler les composants applicatifs produits pendant la phase de génération de code, les composants intergiciels configurés lors de la phase de déploiement et de configuration et les composants fournis par l'utilisateur. Cette édition des liens permet de produire le ou les exécutables correspondant aux nœuds de l'application. Le nombre des exécutables peut être supérieur à un dans le cas de systèmes répartis notamment dans le cas du pacemaker. A l'issue de cette phase, nous disposons d'une application prête à être exécutée.

Dans la suite de ce chapitre, nous détaillons les étapes 2, 3 et 4. Dans les sections 14.2 et 14.3, nous donnons les règles de transformation des modèles AADL pour produire respectivement les composants applicatifs et intergiciels. La section 14.4 décrit la manière dont les composants intergiciels sont déployés et configurés. Dans la section 14.5, nous présentons comment tous les composants logiciels sont intégrés automatiquement pour former une application répartie prête à être exécutée. Enfin, la section 14.6 conclut le chapitre.

14.2. Génération de composants applicatifs

Dans cette partie, nous donnons les règles de transformation des différents composants AADL vers des constructions d'un langage de programmation impératif. Nos règles de transformation sont générales bien que les exemples qui les illustrent sont écrits en Ada. Chacune des règles de transformation est suivie de quelques exemples d'entités AADL et de leurs transformées respectives.

Il faut noter ici que dans certains cas les différences entre les langages d'application générés ne sont pas très grandes. Par exemple, si nous désirons définir un type entier de taille 32 bits dans un langage Ada, le code suivant sera produit :

```
type <Nom_Du_Type> is new Interfaces.Integer_32;
```

probleme LaTeX ici, ajouter
un vskip ?

Tandis que pour le langage C, nous aurons la construction suivante :

```
typedef int_32 <Nom_Du_Type>;
```

Dans d'autres cas, le code Ada est fondamentalement différent du code C. Par exemple, si nous voulons créer un fil d'exécution qui effectue, à chaque seconde, l'appel à un sous-programme **Hello**, en Ada nous aurons le code suivant :

```

task <Nom_Du_Thread> is
    pragma Priority (...);
end <Nom_Du_Thread>;
4   // Création du fil d'exécution
task body <Nom_Du_Thread> is
    Period      : constant Ada.Real_Time.Time_Span := ...;
    Next_Activation : Ada.Real_Time.Time;
9   begin
        Next_Activation := Ada.Real_Time.Clock + Period;
        loop
            Hello;
            delay until Next_Activation;
14        Next_Activation := Next_Activation + Period
        end loop;
    end <Nom_Du_Thread>;

```

En revanche, le langage C ne supporte pas les tâches de manière intrinsèque. Il faut encapsuler le comportement périodique dans un sous-programme et passer par des appels à des bibliothèques comme POSIX pour créer une tâche qui exécute le sous-programme. Nous voyons, dans l'exemple ci-dessous, que l'absence de ce support intrinsèque rend le code C beaucoup plus compliqué que le code en Ada.

```

pthread_mutex_t mutex;
pthread_cond_t cond;

4 void delay_until (struct timespec *next_activation) {
    pthread_mutex_lock (&mutex);
    pthread_cond_timedwait (&cond, &mutex, &next_activation);
    pthread_mutex_unlock (&mutex);
}

9 void wrapper (void) {
    struct timespec period;
    struct timeval next_activation;

14    pthread_mutex_init (&mutex, NULL);
    pthread_cond_init (&cond, NULL);
    period.tv_sec = ...;
    period.tv_nsec = ...;
    gettimeofday (&next_activation);
19    next_activation.tv_sec = next_activation.tv_sec + period.ts;
    next_activation.tv_nsec = next_activation.tv_usec * 1000 + period.tv_nsec;

    while (1) {
        hello ();
        delay_until (&next_activation);
        next_activation.tv_sec = next_activation.tv_sec + period.ts;
        next_activation.tv_nsec = next_activation.tv_nsec + period.tv_nsec;
    }
}

24 /* Création du fil d'exécution */
pthread_create (&tid, NULL, wrapper, NULL);

```

14.2.1. Transformation des données

Les types de données sont déduits des instances de composants AADL de type **data**. Ces composants sont instanciés à plusieurs endroits du modèle :

- dans les éléments d’interface de type **data** ou **event data** qui se trouvent dans les processus, les tâches ou encore les sous-programmes ;
- dans les paramètres des sous-programmes ;
- lors de la déclaration de variables partagées entre plusieurs entités du modèle.

Ces déclarations sont transformées en définitions de types. Les types définis sont utilisés pour définir les différentes instances. Le caractère asynchrone que nous avons imposé aux communications nous impose de définir une valeur « par défaut » pour chacun de ces types de données définis. Cette valeur est utilisée au moment où les données ne sont pas encore disponibles (à l’initialisation de l’application par exemple ou lors d’une panne partielle qui rend la communication momentanément indisponible).

Ainsi, une déclaration d’une constante de ce type ayant une valeur « par défaut » pour ce type est produite pour chaque déclaration de composant **data**. Ce mécanisme est utile uniquement dans les tâches périodiques et hybrides, seuls processus pouvant être réveillés par un événement temporel sans avoir reçu de donnée valide.

Par convention, le nom de ce type est le même que le nom du composant AADL sauf si ce dernier ne respecte pas les règles de nommage avec le langage ; auquel cas, les instances de projection doivent fournir des moyens pour convertir les noms des composants en noms compatibles avec le langage de programmation (similaires aux règles de projection de l’IDL CORBA vers les différents langages de programmation). Par exemple, le langage Ada interdit de définir des identificateurs qui contiennent des occurrences successives du caractère « _ ». Les identificateurs du modèle AADL de l’utilisateur ne respectant pas cette restriction sont modifiés en insérant un caractère différent de « _ » entre deux occurrences successives de ce dernier.

```

data Time
properties
  Data_Model :: Data_Representation => Integer;
end Time;

data Array_Of_Time
properties
  Data_Model :: Data_Representation => Array;
  Data_Model :: Dimension => (10);
  Data_Model :: Base_Type => (data Time);
end Array_Of_Time ;

```

Listing 14.1 – Types de données en AADL

Le listing 14.1 donne un exemple en AADL d’un type de donnée entier ainsi qu’un type tableau contenant dix entiers. Le listing 14.2 décrit une transformation en langage Ada de ces deux composants. En l’absence d’informations supplémentaires de la part de l’utilisateur (taille du type entier, intervalle, etc.), le type est simplement

```

— Transformation du composant AADL Time.
type Time is new Float;

4 — Transformation du composant AADL Array_Of_Time .
type Array_Of_Time is array (1 .. 10) of Time;

```

Listing 14.2 – Code Ada généré

une extension du type standard **Integer**. L'utilisateur précise davantage la définition d'un type en utilisant les propriétés fournies dans l'ensemble de propriétés prédéclaré **Data_Model**. Nous avons défini cet ensemble de propriétés pour pallier le manque de spécification dans la première version du standard AADL. Cet ensemble fera partie des fichiers de propriétés « annexes » de la prochaine version AADLv2.

La nature du type défini dépend fortement de la valeur de la propriété **Data_Model : :Data_Representation**. Si le composant AADL représente une structure de donnée, il sera transformé en un type enregistrement avec des champs dont les types correspondent aux projections de leurs déclarations respectives.

En cas de présence de sous-programmes accesseurs, leurs implantations doivent être conformes au protocole de gestion de concurrence du composant. Nous voulons que l'accès aux données soit protégé contre la concurrence et que l'inversion de priorité due à cette protection soit bornée. Nous voulons aussi garantir l'absence d'interblocage que pourrait causer cette protection. Une solution consiste à utiliser un protocole d'héritage de priorités. Le profil Ravenscar [BUR 04] recommande l'utilisation du protocole de plafonnement de priorité PCP [SHA 90]. En effet, non seulement il garantit l'absence d'interblocage mais il réduit également le temps de blocage d'une tâche (causé par des blocages successifs).

Peu de langages de programmation impératifs (comme Ada) disposent de manière intrinsèque des constructions nécessaires pour ces variables protégées. Pour les autres langages de programmation (comme le langage C), nous utilisons des bibliothèques système pour créer et initialiser les verrous logiciels nécessaires à la protection des données. Ceci rend le code C beaucoup plus compliqué que le code Ada notamment en termes de vérification et de certification.

Le listing 14.3 donne un exemple d'un type de structure de données modélisé en AADL. Il s'agit d'une structure de trois entiers (représentant une coordonnée dans l'espace). La constante par défaut générée pour cette structure est une structure ayant pour chaque champ la valeur par défaut associée à son type. Le listing 14.4 montre la génération en Ada d'une telle constante.

```

data Time extends
properties
  Data_Model :: Data_Representation => Integer;
end Time;

5   — Une structure de donnée contenant deux Entiers
data Record_Type
properties
  Data_Model :: Data_Representation => Struct;
end Record_Type;

10  data implementation Record_TypeImpl
subcomponents
  X : data Time;
  Y : data Time;
end Record_TypeImpl;

```

Listing 14.3 – Exemples de types AADL

```

— Valeur par défaut pour le type Component_Type.
Time_Default_Value : constant Time := 0.0;

4   — Valeur par défaut pour le type Record_Type_Impl.
Record_Type_Impl_Default_Value : constant Record_Type_Impl :=
  (X => Time_Default_Value,
  Y => Time_Default_Value);

```

Listing 14.4 – Code Ada généré

14.2.2. Transformation des sous-programmes

Les sous-programmes AADL sont des entités destinées à abriter le code de l'utilisateur. La clause FEATURES contient éventuellement des paramètres pour échanger les données avec l'extérieur. En AADL, les sous-programmes sont des entités passives qui doivent être appelées par des tâches pour s'exécuter. Ainsi, les composants sous-programmes se transforment naturellement en sous-programmes du langage de programmation impératif choisi pour la génération de code.

Le comportement du sous-programme produit dépend de la nature du composant AADL. Nous classons les sous-programmes AADL en trois catégories en fonction des valeurs des propriétés standards Source_Language et de la présence ou non de la clause CALLS dans l'implantation du composant. Il s'agit des sous-programmes opaques implantés dans un langage de programmation particulier, des sous-programmes à séquence pure d'appel et des sous-programmes vides.

Grâce à la présence de port en sorties (*out*), les sous-programmes AADL (toutes catégories confondues) peuvent déclencher des événements et des événements-données du côté de l'appelant. Ceci permet de contrôler l'envoi d'événements depuis le code de l'utilisateur en utilisant les méthodes définies par le standard AADL pour déclencher les événements. La transformée de tels sous-programmes doit prendre en compte le

fait qu'un même sous-programme (ayant des ports en sortie) est invoqué par plusieurs tâches sur le même nœud. Autrement dit, cette transformation doit garantir que les données ne seront pas corrompus par des accès concurrents.

Nous avons choisi d'utiliser un paramètre « opaque » additionnel qui représente l'état des ports d'un sous-programme à chaque appel de manière indépendante. Ceci garantit ainsi la cohérence des données en cas de concurrence. Nous reviendrons plus en détail sur cette catégorie de sous-programmes lors de la transformation des tâches. En effet, le code généré dans les deux cas est très similaire.

Le générateur de code déduit la catégorie d'un sous-programme à partir de ses caractéristiques. Ensuite selon la catégorie trouvée, le générateur produit le code correspondant. Dans la suite, nous expliquons pour chaque catégorie de sous-programme le comportement du générateur.

Transformation des sous-programmes opaques. Il s'agit de l'implantation la plus immédiate d'un sous-programme en AADL. En effet, toute la partie comportementale est à la charge de l'utilisateur et le modèle AADL contient uniquement la définition de l'interface du sous-programme. Pour ces sous-programmes, les trois propriétés Source_Language, Source_Name et Source_Text sont définies et la clause CALLS est absente. Le modèle AADL indique le langage de programmation, le nom et le fichier source de son implantation. Ces composants donnent lieu à la génération d'un sous-programme squelette qui effectue un appel à l'implantation indiquée. Dans le cas de compilateurs comme le compilateur Ada GNAT où le nom d'un fichier est déduit du nom de l'unité de compilation qu'il contient, le renseignement du champ Source_Text est optionnel.

```

subprogram Battery_Test
features
3   Level : out parameter Base_Types :: Float_32;

properties
  source_language => Ada95;
  source_name      => "Pacemaker.Do_Battery_Test";
8 end Battery_Test;
```

Listing 14.5 – Sous-programme opaque

```

1   with Pacemake;
2   procedure Battery_Test (Level : out Base_Types.Float_32)
      renames Pacemaker.Do_Battery_Test;
```

Listing 14.6 – Code Ada généré

Le listing 14.5 montre le modèle AADL d'un sous-programme opaque implanté en Ada. Ce sous-programme sera transformé en une enveloppe de code qui appelle l'implantation **Ping.Do_Ping_Spg** fournie par l'utilisateur comme l'indique l'extrait de code 14.6.

Transformation des sous-programmes à séquence pure d'appel. Dans ce cas, le sous-programme contient une et une seule séquence d'appel dans sa clause CALLS. Aucune des trois propriétés indiquées plus haut ne doit être définie pour ce type de sous-programme. Le comportement du sous-programme est donc complètement défini par son modèle AADL : une suite d'appels à d'autres sous-programmes. Ce type de composant donne lieu à la génération d'un sous-programme qui effectue la séquence d'appels vers les transformées respectives des composants appelés. Bien entendu, il est possible de connecter les paramètres en entrée et en sortie des sous-programmes appelés. Le code généré doit fournir toutes les variables intermédiaires nécessaires pour garantir le bon flux de données spécifié par ces connexions.

```

2   subprogram Battery_Test
3     features
4       Level: out parameter Base_Types::Float_32;
5       properties — ...
6     end Battery_Test;
7
7   subprogram Internal_Transmitter
8     features
9       Level : in parameter Base_Types::Float_32;
10      Output : out parameter Base_Types::Float_32;
11      properties — ...
12    end Internal_transmitter;
13
13   subprogram Sender
14     features
15       Level : out parameter Base_Types::Float_32;
16     end Sender;
17   subprogram implementation Sender.impl calls {
18     Test : subprogram Battery_Test;
19     Transmit : subprogram Internal_Transmitter;};
20   connections
21     C_1 : parameter Test.Level --> Transmit.Input;
22     C_2 : parameter Transmit.Output --> Level;
23   end Sender.impl;
```

Listing 14.7 – Sous-programme à séquence d'appel

Le listing 14.7 présente un sous-programme AADL **Sender.impl** dont le comportement consiste à appeler le sous-programme **Internal_Sender** puis le sous-programme **Internal_Transmitter**. On ne s'intéresse pas à la manière dont ces deux derniers sont implantés. Le listing 14.8 correspond au code Ada généré à partir de ce modèle. Le sous-programme **Sender.impl** donne lieu à un sous-programme appelé **SenderImpl**. Celui-ci appelle les deux sous-programmes produits à partir de **Internal_Sender** et **Internal_Transmitter**.

```

1  procedure Battery_Test (Level : out Float) is
2      ...
3  end Battery_Test;

4  procedure Internal_Transmitter
5      (Level : Float;
6      Output : out Float)
7  is
8      ...
9  end Internal_Transmitter;

10 procedure Sender_Impl (Output : out Float) is
11     C_1_Var : Float; — Corresponds to the C_1 connection.
12 begin
13     Internal_Sender (C_1_Var);
14     Internal_Transmitter (C_1_Var, Output);
15 end Sender_Impl;

```

Listing 14.8 – Code Ada généré

Le générateur de code traite automatiquement la génération de variables intermédiaires correspondant aux connexions liant les sous-programmes appelés. Dans ce cas, une variable intermédiaire est générée : **C_1_Var**. Elle correspond à la connexion **C_1** qui lie le paramètre en sortie **Output** de **Internal_Sender** au paramètre en entrée **Input** de **Internal_Transmitter**.

Transformation des sous-programmes vides. Les sous-programmes vides sont des sous-programmes pour lesquels l'utilisateur n'a spécifié aucune caractéristique comportementale. Ils donnent lieu à la production d'un sous-programme qui lève une erreur fatale à l'exécution. Ce genre de sous-programme paraît inutile au premier abord. Cependant, nous les utilisons pour tester les modèles lors des premières phases du développement. Après, nous « remplissons » ces sous-programmes en spécifiant leurs comportements respectifs.

14.2.3. Transformation des fils d'exécution

Les composants **thread** sont transformés en des constructions du langage qui permettent de construire une tâche. Peu de langages de programmation offrent des constructions intrinsèques pour créer de tels fils d'exécution (comme le langage Ada par l'intermédiaire du mécanisme de tâches). D'autres langages ont recours à des appels à des bibliothèques systèmes pour créer ces fils d'exécution (comme POSIX [IEE 94] utilisée par des nombreux langages).

Comme nous l'avons précisé, seuls trois types de tâches sont acceptés pour garantir une analyse d'ordonnançabilité statique de l'application (périodique, sporadique et hybride). Pour chacun de ces trois types, un patron de conception est construit dans le langage impératif choisi. Il décrit le comportement de la tâche d'après sa catégorie mais aussi d'après la sémantique que le standard AADL associe aux composants

thread [SAE 04, section 5.3]. Ce patron doit être instancié pour chaque tâche dans l’application en fonction des caractéristiques et des propriétés du composant **thread** qui lui correspondent. Les paramètres nécessaires à l’instanciation de ce patron sont :

- la liste des éléments d’interface du composant. Cette liste est requise pour les threads sporadiques et hybrides ;
- l’identificateur du *thread* ;
- la période pour les tâches périodiques et hybrides ou le temps minimal entre deux déclenchements pour les tâches sporadiques ;
- l’échéance ;
- la priorité ;
- la taille de la pile ;
- pour les tâches sporadiques et hybrides, un sous-programme qui effectue l’attente d’un événement extérieur. Un appel à ce sous-programme bloque la tâche jusqu’à l’arrivée d’un événement et retourne l’élément d’interface qui l’a reçu ;
- le sous-programme qui effectue le travail cyclique ;
- un éventuel sous-programme qui initialise le *thread*.

Certains paramètres sont déduits directement du modèle AADL par simple lecture de propriétés (période, échéance, priorité¹, taille de la pile et sous-programme servant à l’initialisation). Les autres paramètres doivent être générés automatiquement en analysant le composant *thread*. Dans la suite, nous donnons pour chacun des constituants d’un composant **thread**, les règles de transformation vers un langage de programmation impératif.

Transformation des éléments d’interface. La liste des éléments d’interface d’un composant **thread** est transformée en une énumération. Nous avons choisi cette transformation pour être capables d’une part de créer des tableaux indexés par les différents ports d’un composant **thread** et d’autre part d’utiliser les énumérateurs dans des structures conditionnelles à choix multiples (*switch case*). Ceci permet un temps constant d’exécution indépendant du nombre de ports dans un composant **thread**.

Le listing 14.9 montre un modèle AADL d’une tâche sporadique ayant un ensemble de ports en entrée et en sortie. Cet ensemble de ports est traduit, en Ada par exemple, par la définition d’un type énuméré comme le montre l’exemple de code.

1. Si l’on utilise RMS pour ordonner un ensemble de tâches, leurs priorités doivent être déduites de leurs périodes respectives. Dans un premier temps, nous supposons que l’utilisateur donne des périodes cohérentes avec RMS dans son modèle AADL et nous effectuons un ordonnancement de type *FIFO within priorities*.

```

thread Rate_Computation
features
3   Activity: in data port acceleration_intensity;
    Regulated_Rate_Delay: out data port PACEMAKER_Interfaces :: pacing_delay;
properties
  Dispatch_Protocol => Periodic;
  Priority => 2;
8  Period => 100 ms;
end Rate_computation;

```

Listing 14.9 – Déclaration d'un *thread* AADL

```

1 type Rate_Computation_Port_Type is
  (Activity, Regulated_Rate_Delay);

```

Listing 14.10 – Code Ada pour les ports du *thread*

Notons que cette même règle de transformation est appliquée aux sous-programmes qui possèdent des ports dans leur interface. Elle donne lieu à une énumération qui sera utilisée par les routines d'envoi d'événements.

Transformation de la partie comportementale. La construction du sous-programme qui effectue le travail cyclique d'un *thread* dépend fortement du comportement spécifié par l'utilisateur. En effet, nous autorisons trois manières différentes pour spécifier le comportement d'un composant **thread**.

Tâches avec une séquence d'appel. Le comportement de ces tâches est assez simple et très similaire à celui des sous-programmes à séquence pure d'appel. Elles exécutent à chaque occurrence la liste des sous-programmes appelés dans leur séquence d'appel. Cette manière d'implanter un composant **thread** est très adaptée aux tâches périodiques. Tout le comportement est géré automatiquement par le générateur de code et l'utilisateur n'a pas à manipuler les ports de la tâche pour envoyer ou recevoir des informations.

Le sous-programme généré est similaire à celui d'un sous-programme AADL qui appelle lui-même d'autres sous-programmes (donné dans les listings 14.7 et 14.8). Le générateur de code produit les variables intermédiaires nécessaires pour la connexion entre les sous-programmes appelés. Il produit la collecte des données reçues dans les ports en entrée du composant **thread** et les appels des sous-programmes dans le bon ordre. Enfin, le générateur produit l'envoi des éventuels données ou événements sur les ports en sortie du composant.

Tâches avec un point d'entrée par port. Dans ce cas chacun des ports événements (ou événement donné) en entrée du composant **thread** se trouve associé à un sous-programme par l'intermédiaire de la propriété **Compute_EntryPoint** appliquée à

chacun des ports. Le composant **thread** décrit dans l'exemple 14.9 fait partie de cette catégorie. Ainsi, à chaque déclenchement de la tâche, le point d'entrée correspondant au port qui a reçu l'ordre de déclenchement est exécuté. Cette manière d'implanter est adaptée aux tâches sporadiques et hybrides.

Pour les tâches hybrides, l'utilisateur doit définir, en plus des points d'entrée des ports, un point d'entrée pour la tâche lui-même pour réaliser la partie périodique de son exécution.

Le générateur produit un sous-programme qui possède un paramètre dénotant le port qui a déclenché la tâche. Ensuite dans le sous-programme produit, un test est effectué sur la valeur de ce port et le point d'appel correspondant est appelé.

```
thread A_Thread
features
  3  Input_1 : in event data port Integer {Compute_EntryPoint =>
    "Pkg.On_Input_1"};
  Input_2 : in data port Integer;
  Input_3 : in event port {Compute_EntryPoint => "Pkg.On_Input_3"};
  Output_1 : out event data port Integer;
  Output_2 : out data port Integer;
  Output_3 : out event port;
8   properties
    Dispatch_Protocol => Sporadic;
    ...
end A_Thread;
```

Listing 14.11 – Implantation d'un *thread* AADL

```
with Pkg;
procedure A_Thread_Job (Port : A_Thread_Port_Type) is
3  begin
    case Port is
      when Input_1 => Pkg.On_Input_1 (Get_Value (Input_1));
      when Input_3 => Pkg.On_Input_2;
      when others => raise Program_Error;
8    end case;
end A_Thread_Job;
```

Listing 14.12 – Le « travail » du *thread* en Ada

Le listing 14.11 reprend le composant **A_Thread** décrit dans le listing 14.9. Il lui ajoute la propriété **Dispatch_Protocol** pour préciser qu'il s'agit d'une tâche sporadique et associe un point d'entrée à chaque port en entrée. Puisque ce composant possède un point d'entrée par port, le comportement du sous-programme généré exécute le point d'entrée correspondant à l'événement reçu. L'utilisation d'énumération pour représenter les ports nous permet de trouver le comportement voulu en temps constant. Comme le montre le listing 14.12, chacun des points d'entrée pour les ports de type événement-donnée possède un paramètre dans sa signature. Ceci permet au

code de l'utilisateur de retrouver la valeur de la donnée reçue par le port. La lecture de la donnée s'effectue à l'aide de la routine **Get_Value**.

Dans le cas des tâches avec un seul point d'entrée, ce dernier est spécifié par l'intermédiaire de la propriété standard **Compute_EntryPoint** associée au composant **thread** lui-même. Il s'agit de permettre à l'utilisateur de gérer lui-même tout le comportement lors d'une occurrence (gestion des files d'attentes des ports, dialogue avec les routines de l'intericiel pour envoyer des informations, etc.).

Cette méthode de conception est utilisée dans des cas rares où l'utilisateur désire effectuer des comportements avancés sur les files d'attentes des éléments d'interface comme l'extraction de plus d'un élément par exemple ou encore la lecture d'une file d'attente différente de celle du port qui vient de provoquer le déclenchement. Dans ce cas, l'utilisateur masque une partie du comportement à la phase de vérification.

Transformation des modes opérationnels. Les modes opérationnels d'un composant **thread**, tels que nous les supportons, ne peuvent contrôler que le comportement de la tâche correspondante. Ainsi, ils ne compromettent ni compliquent l'analysabilité statique de l'application. Dans ce cas, d'une manière similaire à celle utilisée pour transformer les éléments d'interface, les modes sont transformés en une énumération. La machine à états qui pilote le changement de mode est transformé en une machine à états dans le langage de programmation impératif. Elle est placée au début du sous-programme généré pour effectuer le comportement cyclique. Ainsi, au début de son exécution, la valeur du mode est calculée en fonction de événements reçus et de la valeur courante. Ensuite le sous-programme appelé effectue le traitement en fonction du mode trouvé.

```

1  thread A_Thread
2   features
3     Go_Lazy : in event port;
4     Go_Normal : in event port;
5     Go_Crazy : in event port;
6   properties Dispatch_Protocol => Sporadic;
7   end A_Thread;
8   thread implementation A_ThreadImpl
9   modes
10    Normal : initial mode;
11    Crazy : mode;
12    Lazy : mode;
13    Normal, Lazy -[Go_Crazy ]-> Crazy;
14    Normal, Crazy -[Go_Lazy ]-> Lazy;
15    Crazy, Lazy -[Go_Normal]-> Normal;
16   properties
17     Compute_EntryPoint => "Pkg.Normal_Handler" in modes (Normal);
18     Compute_EntryPoint => "Pkg.Crazy_Handler" in modes (Crazy);
19     Compute_EntryPoint => "Pkg.Lazy_Handler" in modes (Lazy);
20   end A_ThreadImpl;
```

Listing 14.13 – Modes dans *thread* AADL

```

type Mode_Type is (Normal, Crazy, Lazy);
Mode : Mode_Type := Normal;
procedure A_Thread_Job (Port : A_Thread_Port_Type) is
begin
  case Port is
    when Work_Normally =>
      case Mode is
        when Crazy | Lazy => Mode := Normal;
        when others => null;
      end case;
    when _ ...
    when others => null;
  end case;
  case Mode is
    when Normal => Pkg.Normal_Handler (Port);
    when Crazy => Pkg.Crazy_Handler (Port);
    when Lazy => Pkg.Lazy_Handler (Port);
  end case;
end A_Thread_Job;

```

Listing 14.14 – Le « travail » du *thread* en Ada

Le listing 14.13 montre un modèle AADL d'un composant **thread** sporadique possédant trois modes opérationnels. Le composant possède un seul point d'entrée. Le changement de mode est décrit par la machine à états donnée dans la clause MODES du composant **A_ThreadImpl**. Selon la valeur du mode courant, la tâche exécute un des sous-programmes spécifiés dans la clause PROPERTIES du même composant.

Le code Ada produit pour gérer le changement de modes est présenté dans le listing 14.14. Nous y remarquons notamment l'énumération représentant les modes (**Mode_Type**). La variable globale représentant le mode courant est initialisée à **Normal** comme précisé dans le modèle AADL. De plus, le code généré implante la machine à état qui pilote le changement de mode et effectue le comportement voulu. Là aussi, l'utilisation d'une énumération pour représenter les modes opérationnels permet d'effectuer les calculs et les transitions en un temps constant.

Instanciation de la tâche. Un composant AADL **thread** est transformé en un couple formé :

- de l'instance Ada du patron de conception pour créer un fil d'exécution ;
- des méthodes qui permettent l'accès en lecture et/ou en écriture à l'interface du composant afin d'envoyer et recevoir les informations.

Le listing 14.15 montre l'instance Ada générée pour **A_ThreadImpl** en Ada. Le générateur de code affecte des valeurs par défaut aux paramètres optionnels qui ne sont pas fournis par l'utilisateur. Ainsi, l'échéance de la tâche prend une valeur égale à celle de la période et sa priorité est fixée à la priorité par défaut du système.

Le sous-programme **Wait_For_Incoming_Events** qui est le dernier paramètre de l'instanciation de **A_ThreadImpl** fait partie des méthodes du service d'interaction

```

1 package A_Thread_Task is new Sporadic_Task
2   (Port_Type          => Receiver_Port_Type,
3    Entity              => SC_2_Receiver_ID,
4    Task_Period         => Ada.Real_Time.Milliseconds (20),
5    Task_Deadline       => Ada.Real_Time.Milliseconds (20),
6    Task_Priority       => System.Default_Priority,
7    Task_Stack_Size     => 64_000,
8    Job                 => A_Thread_Job,
9    Wait_For_Incoming_Events => Wait_For_Incoming_Events);

```

Listing 14.15 – Instance d'un *thread* sporadique en Ada

produit automatiquement. Ce service constitue une couche au-dessus du service d'interaction de l'intergiciel minimal. Il fournit les routines suivantes qui permettent d'accéder en lecture et en écriture aux éléments d'interface d'un composant **thread**.

Les noms de ces routines ainsi que leurs comportements sont spécifiés par le standard AADL :

- **Send_Output** envoie explicitement les événements et les données à travers le réseau s'il s'agit d'une communication distante. Si la communication est locale, cette méthode effectue la copie directe vers la destination. Elle prend en paramètre une référence vers le port à traiter ;
- **Put_Value** marque un port comme contenant une information (donnée ou événement) prête à être envoyée. L'envoi de la donnée sera effectué par un appel à **Send_Output**. Elle prend comme paramètres une référence vers le port en question ainsi que la valeur de la donnée à envoyer s'il s'agit d'un port donnée ou événement-donnée ;
- **Get_Count** retourne le nombre de messages non consommés dans la file d'attente d'un port événement ou événement-donnée ;
- **Get_Value** renvoie la valeur reçue présente en tête de la file d'attente d'un port événement ou événement-donnée ou tout simplement la valeur d'un port de type donnée. Elle ne provoque pas de consommation : plusieurs appels successifs à cette méthode retournent le même résultat ;
- **Next_Value** consomme le message à la tête de la file d'attente d'un port événement ou événement-donnée ;
- **Wait_For_Incoming_Events** concerne les tâches sporadiques. Elle attend jusqu'à l'arrivée d'événements, et retourne le port sur lequel l'événement est arrivé.

Le comportement étant générique est faiblement personnalisable en fonction des propriétés d'une application, elles sont rassemblées sous forme d'un archétype qui fait partie de l'intergiciel minimal. Il s'agit du service d'interrogation de l'intergiciel minimal. Pour chaque tâche, une instance Ada de ces routines est créée. Nous créons une instance Ada pour chaque composant AADL afin de pouvoir la paramétriser statiquement.

L’instance Ada est paramétrée par les entités suivantes :

- un type énuméré qui liste les ports du composant **thread** ;
- un type structure de donnée variable (comme les unions en C) qui est piloté par le type énuméré cité plus haut. Pour chaque port donnée ou événement donnée, cette structure contiendra un champ supplémentaire ayant le type de la donnée. Pour les ports de type événement pur, cette structure est vide ;
- un identificateur pour le *thread* ;
- une table indiquant pour chaque port, sa catégorie et son sens de propagation (*in data port, out data port, in event port, etc.*) ;
- une table indiquant pour chaque port événement en entrée, la taille de sa file d’attente (spécifiée à l’aide de la propriété AADL « Queue_Size ») ;
- une table indiquant pour chaque port en sortie, la liste de ses destinations ;
- une routine *Marshall* qui emballle une information de type *interface du thread* dans une file d’attente ;
- une routine *Next_Deadline* qui retourne la date de la prochaine échéance du *thread*.

Le listing 14.16 montre comment ces interrogateurs sont instanciés dans le cas de Ada par exemple. Des entités supplémentaires doivent être fournies au paquetage générique pour compléter l’instanciation comme les différents types. Le nouveau paquetage instancié **A_Thread_Interrogators** fournit les sous-programmes cités plus haut.

Pour les sous-programmes AADL possédant des ports, un ensemble d’interrogateurs est créé de manière similaire.

14.2.4. Transformation des instances de données partagées

Les instances des données protégées sont les sous-composants de types **data** qui sont déclarés dans les implantations des composants **process**. Il s’agit de données partagées entre les tâches d’un même processus.

Un sous-composant **data** est caractérisé par son nom et le type du composant qu’il instancie. Il est transformé en une déclaration de variable partagée (protégée ou non, selon la spécification de l’utilisateur). Un ensemble de méthodes prévues pour accéder à cette variable est généré. Ces méthodes sont utilisées par les tâches ou les sous-programmes partageant l’accès à cette variable.

```

1  type Rate_Computation_Interface (Port : Rate_Computation_Port_Type) is
2    record
3      case Port is
4          when Activity           => Activity_DATA : Integer;
5          when Regulated_Rate_Delay => Regulated_Rate_Delay_DATA : Integer;
6      end case;
7    end record;
8
9    type Rate_Computation_Integer_Array is
10       array (Rate_Computation_Port_Type) of Integer;
11    type Rate_Computation_Port_Kind_Array is
12       array (Rate_Computation_Port_Type) of Port_Kind;
13       — Des types pour définir les tableaux ci-dessous
14
15    Rate_Computation_Port_Kinds : constant Rate_Computation_Port_Kind_Array :=
16        (Activity           => In_Event_Data_Port ,
17         Regulated_Rate_Delay => Out_Event_Data_Port);
18
19    Rate_Computation_FIFO_Sizes : constant Rate_Computation_Integer_Array :=
20        (Activity           => 16, — Valeur par défaut pour un port d'évènement
21         Regulated_Rate_Delay => -1); — Valeur conventionnelle (port en sortie)
22
23    type Rate_Computation_Address_Array is
24       array (Rate_Computation_Port_Type) of System.Address;
25
26    Rate_Computation_N_Destinations : constant Rate_Computation_Integer_Array :=
27        (Activity           => -1, — Valeur conventionnelle (port en entrée)
28         Regulated_Rate_Delay => <Nbr Destinations>); — Déduite de la topologie
29
30    Rate_Computation_Destinations : constant Rate_Computation_Address_Array :=
31        (Activity           => System.Null_Address ,
32         Regulated_Rate_Delay => <Tableau Des Ports Destination>'Address);
33       — Instance du paquetage générique
34
35    package Rate_Computation_Interrogators is new Thread_Interrogators
36        (Port_Type           => Rate_Computation_Port_Type ,
37         Integer_Array       => Rate_Computation_Integer_Array ,
38         Port_Kind_Array    => Rate_Computation_Port_Kind_Array ,
39         Address_Array       => Rate_Computation_Address_Array ,
40         Thread_Interface_Type => Rate_Computation_Interface ,
41         Current_Entity     => Rate_Computation_Impl_ID ,
42         Thread_Port_Kinds  => Rate_Computation_Port_Kinds ,
43         Thread_Fifo_Sizes  => Rate_Computation_FIFO_Sizes ,
44         N_Destinations     => Rate_Computation_N_Destinations ,
45         Destinations        => Rate_Computation_Destinations ,
46         Marshall           => Marshall ,
47         Next_Deadline       => Rate_Computation_Task.Next_Deadline );

```

Listing 14.16 – Instance Ada des «interrogateurs» pour **A_Thread_Impl**

14.3. Génération de composants intergiciels

Comme précisé précédemment, une partie de l'intergiciel est produite automatiquement. Il s'agit des composants restants des services intergiciels fortement personnalisables. Ces composants sont dédiés aux caractéristiques du nœud en cours. Il s'agit des composants suivants :

– les instances Ada de *threads* : pour chaque composant AADL de type **thread**, une instance du paquetage Ada générique correspondant à sa catégorie est créée. Toutes ces instances sont créées dans le paquetage **PolyORB_HI_Generated.-Activity** ;

– les routines d’emballage et de déballage avancées correspondant aux types de données présents dans le modèle AADL. Il s’agit de sous-programmes utilisant des instances du paquetage Ada générique **PolyORB_HI.Marshallers_G** appartenant à l’intergiciel minimal. Ce paquetage offre les mécanismes de représentation élémentaire. Les sous-programmes générés implantent le service de représentation avancée. Ils appartiennent au paquetage **PolyORB_HI_Generated.Marshallers** ;

– les tables statiques de nommage correspondant à chacune des services de transport supportés. Ces tables sont produites dans le paquetage **PolyORB_HI_Generated.Naming** ;

– le service de transport de haut niveau dont les routines permettent de sélectionner correctement le service de transport de bas niveau selon la source et la destination d’un message. Elle est produite dans le paquetage **PolyORB_HI_Generated.Transport**.

14.4. Déploiement et configuration des composants intergiciels

Dans cette section, nous expliquons comment sont sélectionnés les composants de l’intergiciel minimal dont l’application a besoin (déploiement). Nous expliquons aussi comment ces composants ainsi que ceux produits automatiquement sont paramétrés en fonction des caractéristiques de l’application (configuration).

14.4.1. Déploiement

La sélection des composants de l’intergiciel minimal se fait lors du parcours de l’arbre d’instance du modèle AADL de l’application. En particulier, les propriétés des composants matériels sont analysées pour déterminer précisément les composants de l’intergiciel minimal dont ils ont besoin. Nous utilisons l’ensemble des propriétés de déploiement que nous avons définies dans [ZAL 08].

Bus. Les propriétés d’un bus AADL permettent de sélectionner les versions nécessaires des services de l’intergiciel minimal :

– le service de transport de bas niveau : en effet, parmi les propriétés additionnelles relatives au déploiement que nous avons ajoutées, une propriété **Transport_API** permet de préciser le mécanisme de bas niveau de transport qu’utilise le bus (Ethernet, SPACEWIRE) ;

– le protocole : en effet, le bus contient aussi des informations sur le protocole utilisé. La propriété **Protocol** que nous avons introduite précise le protocole utilisé pour échanger les données à travers les bus ;

– la représentation élémentaire : généralement, chaque protocole utilisé est associé à un service de représentation qui indique la manière selon laquelle les données seront échangées. Par exemple, le protocole par défaut qui consiste à envoyer les données dans le réseau par une simple copie ne nécessite pas un service de représentation évoluée. Il suffit de convertir les données en des suites d'octets sans les modifier. En revanche, pour un protocole comme GIOP, nous utilisons la représentation CDR spécifiée par CORBA ;

Processeurs. Les processeurs précisent la plate-forme sur laquelle le nœud s'exécute. Ceci permet de raffiner la sélection des services. Par exemple, si un même service est implanté différemment pour deux plates-formes différentes, la nature du processeur précise la version à choisir. La plate-forme d'un processeur est spécifiée à l'aide de la propriété **Execution_Platform** que nous avons introduite.

Configuration. La configuration des services de l'intergiciel est effectuée statiquement et très simplement grâce aux différentes énumérations et constantes que nous produisons en transformant les entités AADL. Dans cette partie, nous donnons ces énumérations et constantes et nous précisons quels services elles configurent.

Nœuds. Sur chaque nœud de l'application, nous produisons une énumération contenant la liste des nœuds auxquels il est connecté. Nous produisons aussi une constante qui a le même type énuméré et qui permet au nœud de s'identifier lui-même.

Cette énumération est utilisée pour configurer le service de transport de bas niveau en ouvrant des canaux de communication uniquement pour les nœuds réellement connectés.

Threads. Sur chaque nœud de l'application, nous produisons une énumération contenant la liste des composants **threads** de ce nœud ainsi que les composants **threads** des autres nœuds qui communiquent avec le nœud courant. Nous produisant aussi une table qui associe ces composants **threads** avec les nœuds auxquels ils appartiennent.

Cette énumération ainsi que cette table sont utilisées pour compléter l'initialisation des services de transport de bas niveau en ouvrant le nombre correct de canaux de communication vers chaque nœud distant.

Taille des messages. Sur chaque nœud, nous analysons toutes les données qui y sont manipulées et nous calculons la taille maximale d'un message envoyé ou reçu. Par la suite, nous générerons une constante qui représente cette taille. Cette constante permet d'initialiser les services de protocole et de représentation élémentaire en allouant statiquement les files d'attente.

Notons que l'évaluation de cette constante ne se base pas uniquement sur la taille des données transmises. Elle prend en compte également la taille des entêtes des messages.

14.5. Intégration de la chaîne de compilation

Après la génération des composants applicatifs et intergiciels de l’application répartie, la dernière phase consiste à intégrer ce code aux composants déjà existants de l’intericiel minimal ainsi qu’à ceux de l’utilisateur. Par la suite, le code est compilé pour avoir un nombre d’exécutables (égal au nombre des nœuds dans l’application) prêts à être téléchargés et exécutés dans leurs emplacements respectifs.

Rendre cette phase automatique paraît simple au premier abord. Mais une analyse poussée du problème montre que cela demande une quantité de travail non négligeable comparée aux autres phases du processus (analyses, génération de code, etc.) :

- certains compilateurs de langage de programmation requièrent des fichiers d’accompagnement des fichiers sources pour compiler ceux-ci (comme le compilateur Ada GNAT et les fichiers projets ou encore les Makefiles) ;
- l’intégration des composants de l’intericiel minimal requiert que le bon compilateur soit sélectionné selon le langage de programmation choisi, que la version du compilateur croisée convenable soit sélectionnée selon l’architecture du nœud donnée dans le modèle de l’application ;
- l’intégration de composants précompilés et/ou produits par des outils tiers requiert l’ajout d’options lors de l’invocation du compilateur.

La figure 14.2 décrit le processus global de compilation. Après avoir servi à la génération des composants applicatifs et intergiciels, le modèle AADL est aussi utilisé pour piloter la phase de configuration de l’intericiel. Cette phase consiste à sélectionner les composants de l’intericiel minimal dont l’application a besoin et à les intégrer aux composants générés automatiquement pour produire un intericiel configuré et dédié à l’application. Ceci est effectué notamment grâce à l’incorporation de la description des composants matériels dans le modèle.

Le modèle AADL donne lieu aussi à la génération de fichiers de supports (Makefiles, fichiers projet, etc.). Ces fichiers pilotent la phase de compilation qui prend en entrée le code de l’utilisateur (y compris le code produit par des outils tiers) ainsi que le code de l’intericiel configuré.

Pour produire ces fichiers, une analyse de tout le modèle de l’application est nécessaire afin de connaître les dépendances entre les noms des fichiers sources, les dépendances entre eux, etc. La partie matérielle du modèle AADL est aussi analysée afin de déduire le compilateur (natif ou croisé) qui sera utilisé. A l’issue de la phase de compilation, nous disposons d’une application répartie prête à être exécutée.

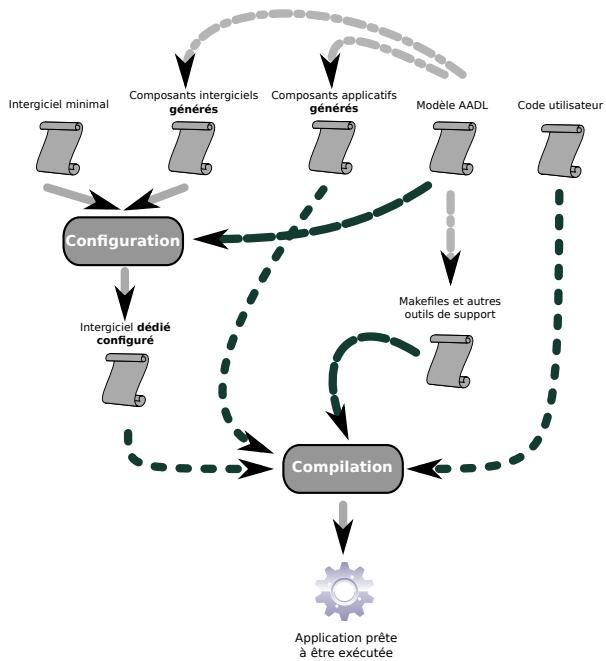


Figure 14.2 – Processus global de compilation

14.6. Conclusion

Dans ce chapitre, nous avons présenté les différentes étapes d'un processus automatique de production de systèmes TR²E critiques à partir de modèles AADL. En partant d'un modèle AADL écrit selon les restrictions que nous avons spécifiées dans le chapitre précédent, nous avons présenté les différentes analyses à effectuer sur ce modèle pour garantir un fonctionnement correct du système. Nous avons montré trois sortes d'analyses qu'il est possible d'effectuer sur un modèle AADL. Elles varient des analyses sémantiques simples aux analyses avancées d'ordonnancement.

La partie la plus importante de ce processus est la génération automatique de code qui, à partir du modèle AADL de l'application, produit les différents composants applicatifs et intergiciels qui seront intégrés avec les composants déjà existants pour former les différents nœuds de l'application. Nous avons présenté les règles de transformation des différents composants AADL vers un langage de programmation impératif générique. Nous avons ensuite montré comment les composants intergiciels sont déployés et configurés en fonction des caractéristiques de l'application TR²E en cours. Notre processus de génération de code produit un code source lisible complètement adapté aux besoins de l'application.

14.7. Bibliographie

- [BUR 04] BURNS A., DOBBING B., VARDANEGA T., « Guide for the use of the Ada Ravenscar Profile in high integrity systems », *Ada Lett.*, vol. 24, n° 2, p. 1-74, ACM, 2004.
- [IEE 94] IEEE I., *IEEE Standard for Information Technology - Portable Operating System Interface (POSIX) : System Application Program Interface (API), Amendment 1 : Realtime Extension (C Language)*, IEEE Std 1003.1b-1993, IEEE Standards Office, New York, 1994.
- [SAE 04] SAE, Architecture Analysis & Design Language (AS5506), septembre 2004, available at www.sae.org.
- [SHA 90] SHA L., RAJKUMAR R., LEHOCZKY J. P., « Priority Inheritance Protocols : An Approach to Real-Time Synchronization », *IEEE Trans. Comput.*, vol. 39, n° 9, p. 1175–1185, IEEE Computer Society, 1990.
- [ZAL 08] ZALILA B., Configuration et déploiement d'applications temps-réel réparties embarquées à l'aide d'un langage de description d'architecture, PhD thesis, Ecole Nationale Supérieure des Télécommunications, novembre 2008.