THÈSE

En vue de l'obtention du

DOCTORAT DE L'UNIVERSITÉ DE TOULOUSE

Délivré par :

Institut Supérieur de l'Aéronautique et de l'Espace (ISAE)

Présentée et soutenue par :
Anthony FERNANDES PIRES

le jeudi 26 juin 2014

Titre :

Amélioration des processus de vérification de programmes par combinaison des méthodes formelles avec l'Ingénierie Dirigée par les Modèles

---

École doctorale et discipline ou spécialité :

ED MITT : Sureté de logiciel et calcul de haute performance

Unité de recherche :

Équipe d'accueil ISAE-ONERA MOIS

Directeur(s) de Thèse :

Mme Virginie WIELS (directrice de thèse)
M. Thomas POLACSEK (co-directeur de thèse)

Jury :

M. Yves LEDRU Université Joseph Fourier - Rapporteur
Mme Mireille BLAY-FORNARINO, Université de Nice - Rapporteur
M. Philippe DHAUSSY, ENSTA Bretagne - Examinateur
Mme Virginie WIELS, ONERA - Directrice de thèse
M. Thomas POLACSEK, ONERA - Co-directeur de thèse
M. Stéphane DUPRAT, Atos - Examinateur

Remerciements

Bonjour à vous chers lecteurs !

Vous commencez à lire cette thèse et je vous en suis gré, mais pour moi, la rédaction de cette partie du manuscrit marque la fin de l'écriture et la fin d'une longue et magnifique aventure durant laquelle j'ai rencontré de nombreuses personnes qui m'ont beaucoup apporté et avec qui j'ai pu partager de très bons moments. C'est pour quoi je me dois de leur rendre hommage à travers ces lignes.

J'aimerais tout d'abord remercier mes encadrants qui m'ont guidé tout au long de ces trois années et qui ont cru en moi. Merci à Thomas POLACSEK. Tu sais que sans toi, je n'aurai jamais atterri à Toulouse et je n'aurai sans doute jamais imaginé faire une thèse. Je te suis immensément reconnaissant pour tout ce que tu as fait pour moi, pour tout ce que tu m'as appris, tous tes bons conseils, toutes les bonnes discussions et tous les bons moments que tu m'as permis de vivre. Merci du fond du cœur. Merci ensuite à Stéphane DUPRAT. Tu m'as également beaucoup apporté grâce à ton immense savoir sur les méthodes formelles et à tes bons conseils. Merci de m'avoir guidé au sein du monde de l'entreprise et surtout merci d'avoir été un si bon et si compréhensif encadrant industriel. Merci à Virginie WIELS. Merci d'avoir toujours trouvé du temps pour suivre mes recherches, d'avoir toujours été là quand j'avais des questions ou besoin d'aide. Enfin merci à Raphaël FAUDOU. Merci pour tes conseils et pour m'avoir fait participer aux projets au sein de l'entreprise.

Je remercie également Yves LEDRU et Mireille BLAY-FORNARINO d'avoir accepté d'être rapporteurs pour ce manuscrit, ainsi que Philippe DHAUSSY, d'avoir accepté de faire partie du jury. Merci d'avoir accordé de votre temps à mon travail, j'en suis très honoré.

Cette thèse n'aurait sans doute jamais commencé sans le soutien de certaines personnes. Je tiens donc à remercier à l'ONERA, Pierre-Loïc GAROCHE, qui a su trouver les bonnes pistes pour la CIFRE, Rémi DELMAS et David DOOSE, qui m'ont soutenu durant le démarrage de la thèse ainsi que Josette BRIAL pour sa bonne humeur et sa parfaite gestion administrative. Je remercie également Jean-Baptiste LAVEDRINE à Atos, ainsi que Patricia FATRAS, qui a toujours été là pour la gestion administrative de cette thèse côté entreprise.

Je tiens ensuite à remercier l'ensemble des personnes avec qui j'ai pu travailler au cours de ces trois dernières années. Tout d'abord l'équipe à Atos : Tristan FAURE, Anne HAUGOMMARD, Mathieu VELTEN, Camille LOUGE, Alain LEFRANCOIS,

Aicha BOUDJELAL, Anass RADOUANI, Olivier MELOIS, Jean-Francois ROLLAND, Philippe ROLAND, Florence VIVARES, Victoria MOYA LAMIEL, Gilles TREMOLIERES, Ivan RAICHS, Yijun CAO, David RABELY, Arthur DAUSSY, Guillaume RENIER, Abdellah EL AYADI, Frederic BARRAILLE, Jean-Baptiste MARCILLE, Michel ABAUZIT, Vincent GUYON, Jêrome DUPEYRON ainsi qu'Alexandre CORTIER, Emilien, Alexia, David, Vincent, Sarah et Aurélie. Merci à vous et à votre bonne humeur ! Travailler à vos côtés et, bien entendu, partager les débats philosophiques de nos pauses-café vont me manquer...

Et puis l'équipe du DTIM évidemment ! : Frédéric BONIOL, Claire PAGETTI, Claire SAUREL, Eric NOULARD, Marc BOYER, David CHEMOUIL, Julien BRUNEL, Guy DURRIEU, Laurence CHOLVY, Olivier POITOU, Pierre BIEBER, Pierre SIRON, Luca SANTINELLI, Jean-Loup BUSSENOT, ainsi qu'Yvonne LE BRETON, Lyne MORON, Bernard LECUSSAN, Pierre MICHEL et bien d'autres. Merci pour votre soutien, pour ce que vous avez pu m'enseigner et pour l'unique et merveilleuse ambiance de travail qui règne au sein du DTIM.

Une pensée particulière pour mes amis doctorants de l'ONERA, mes compagnons galériens, anciens et actuels ! Merci à Florian, Jean-Baptiste, William, Stéphanie, Mikel, Maria, Pierre, Rémy, Christophe, Antoine, Alexandra, Konstantinos, Marc, Tomasz, Vincent, Matthias, Asma, etc.

Merci également aux personnes avec qui j'ai eu l'occasion de travailler au cours de différents projets : Cédrik BESSEYRE, Jack BERINGUIER, Liaiss MERZOUGUE, Jean-Marc CADET, Eric JENN, Mohamed Habib ESSOUSSI, Claudine CHAMONTIN, etc.

Et puis un grand merci à Christophe GARION, qui m'a fait confiance et qui m'a permis d'encadrer des tps de Java à l'ISAE.

Mais ces trois années de dur labeur n'ont pas été uniquement marquées par le travail, mais également par des rencontres avec des gens formidables hors du laboratoire ou de l'entreprise. Je tiens à remercier mon club de karaté, Saint-Sernin Karaté, et tout particulièrement Patrick, le sensei, mais également Mourad, Nico, Antonin et Pauline pour tout ce que vous m'avez enseigné et l'équilibre que vous m'avez apporté au cours de ces trois années de thèse. Je remercie également Laurent, Olivier, Marlène, Pauline, David, Catalina, Sandra et tout les autres pour les bons moments et les dures heures d'entraînement que nous avons pu passer ensemble. Je remercie également Pierre, mon prof de conduite, qui m'a permis de pratiquer la moto durant mon temps libre dans les meilleures conditions possibles pour finalement aboutir sur mon permis à la fin de ma thèse.

Je tiens également à remercier l'ensemble des enseignants qui m'ont enseigné jusque là ainsi que l'ensemble des équipes administratives qui s'occupent des thèses à l'école doctorale MITT et à l'ISAE. Je remercie aussi Atos, l'ONERA et l'ANR pour avoir permis cette thèse.

Enfin, sans doute le plus important, je remercie ma famille sans qui rien de tout ça n'aurait été possible. Tout d'abord mes parents, Antonio et Dina, qui m'ont toujours soutenu dans mes projets et qui ont toujours cru en moi. Merci à vous deux, si j'en suis

là aujourd'hui c'est grâce à vous. Et puis ma soeur, Jamie et sa moitié, Loic, pour leur soutien et leurs attentions durant ma thèse. Je remercie également le reste de ma famille, et plus particulièrement ma grand-mère Idalina pour ses bons conseils. J'ajouterai une pensée particulière pour Jaime et Alessandro.

Ces remerciements touchent maintenant à leur fin. Si vous estimez que je vous ai oublié dans ces lignes, vous pouvez certes commencer par me blâmer pour ce monstrueux oubli, mais vous devez surtout me contacter afin que je vous rajoute au plus vite dans la version électronique. Il y aura toujours de la place pour vous dans mon manuscrit !

Merci à tous !

Cette thèse est dédiée à ma grand-mère,
Martina LOBATI

Table des matières

Chapitre 1

Introduction

Au cours d'un développement logiciel, les activités de vérification sont cruciales étant donné qu'elles permettent de s'assurer de la fiabilité et de la qualité du logiciel produit. C'est pourquoi elles représentent un coût important de tout projet. Ainsi dès les années 60, [76] rapportait que plus de la moitié du temps de développement d'un programme était consacré aux tests et à la correction des erreurs. Plus récemment en 2006, [81] indiquait que la phase de tests pouvait facilement atteindre plus de la moitié des coûts totaux engagés dans un projet de développement logiciel. Aujourd'hui, cette tâche est d'autant plus cruciale que les logiciels sont embarqués dans la plupart des objets qui nous entourent, allant d'un simple téléphone jusqu'aux moyens de transport, que cela soit une voiture, un bus ou bien un avion. Ainsi, le besoin de s'assurer qu'un programme ne contient aucune erreur n'a jamais été aussi grand, puisqu'une erreur de programmation peut avoir, par exemple, une incidence sur la sécurité de données personnelles ou plus directement sur la vie d'usagers. Dans le contexte particulier de ces logiciels embarqués, si un rapport du ministère de l'industrie [107] indique que 40 à 50% des coûts totaux de développement sont dédiés aux activités de vérification et validation, nous avons mesuré chez Atos que les activités de vérification atteignent parfois 60% de la charge de travail dans certains projets [52, 49].

Une des pistes prometteuses pour la réduction de ces coûts de vérification est l'utilisation des méthodes formelles. Ces méthodes s'appuient sur des fondements mathématiques et permettent d'effectuer des tâches de vérification à forte valeur ajoutée au cours du développement. Elles permettent, par exemple, de prouver formellement l'absence d'erreurs dans un programme. Les méthodes formelles sont déjà utilisées dans l'industrie [121, 101, 20, 21, 92]. Cependant, leur difficulté d'appréhension et la nécessité d'expertise pour leur mise en pratique sont un frein à leur utilisation massive.

Parallèlement au problème des coûts liés à la vérification, vient se greffer la complexification des logiciels et du contexte de développement. Aujourd'hui, un logiciel n'est plus développé par une seule entreprise, mais par un ensemble de partenaires qui ont, dans le cadre de projets, des objectifs communs et qui partagent des ressources et des connaissances. Parce qu'il peut être difficile de communiquer entre différents partenaires, il devient essentiel d'offrir un cadre pour que des équipes multiculturelles puissent parta-

ger, discuter et travailler ensemble. De surcroît, les phases amont de spécification sont décisives dans un développement logiciel. Or, l'expression d'une spécification logicielle en langage naturel est parfois source d'ambiguïtés [89] et est donc aujourd'hui mal adaptée au développement de logiciels à la complexité croissante ou à un développement collaboratif. L'Ingénierie Dirigée par les Modèles (IDM) permet de faire face à ces difficultés en proposant des modèles moins ambigüs qu'une description en langage naturel mais également des activités pour automatiquement tirer profit de ces modèles, telles que la génération de code et de documentation.

A la lumière de ces constats, nous avons décidé d'étudier au travers de cette thèse CIFRE [1] réalisée au sein de la société Atos [2] et de l'ONERA [3], les possibilités d'amélioration des processus de vérification de programmes par combinaison des méthodes formelles avec les techniques développées en IDM. L'objectif principal est de permettre aux équipes de développement logiciel, et plus spécifiquement de logiciels embarqués, de bénéficier a minima des atouts des méthodes formelles, sans pour autant changer leur façon de travailler. Les travaux réalisés dans cette thèse prennent évidemment en compte l'état de l'art des différentes thématiques abordées mais également le contexte industriel et technique propre à la société Atos. Dans ce cadre, ces travaux s'intéressent particulièrement aux logiciels embarqués avioniques et aux standards de certification encadrant leur développement. Ils sont également influencés par les techniques de modélisation et les méthodes formelles employées par Atos.

Cette thèse est présentée de la manière suivante. Un état de l'art de l'Ingénierie Dirigée par les Modèles, de la représentation comportementale de logiciels et des techniques de vérification de programmes est présenté Chapitre 2. Le contexte industriel est ensuite décrit dans le Chapitre 3. L'axe de recherche suivi en est déduit et y est défini en détails. Le Chapitre 4 décrit les contributions pour la modélisation de logiciels embarqués. Le Chapitre 5 présente les contributions pour l'implémentation et la vérification de ces logiciels. Le Chapitre 6 présente l'outil réalisé et l'évaluation des contributions par rapport à l'objectif de la thèse, au contexte avionique, aux attentes de la communauté du génie logiciel et par rapport aux travaux connexes. Enfin le Chapitre 7 présente les perspectives envisagées et conclut cette thèse.

1. Conventions Industrielles de Formation par la REcherche
2. `fr.atos.net`
3. Office National d'Etudes et de Recherches Aérospatiales, `www.onera.fr`

Chapitre 2

Etat de l'art

2.1 L'Ingénierie Dirigée par les Modèles

S'abstraire de la réalité pour pouvoir représenter des objets, des concepts ou des problèmes est un procédé que l'être humain utilise naturellement dans la vie de tous les jours. Par exemple, si nous demandons à quelqu'un de représenter un avion, il dessinera le plus souvent un fuselage, des ailes, une dérive, une gouverne de profondeur, etc. En fait, il décrira l'abstraction qu'il se fait d'un avion, en d'autres termes son modèle. En matière d'ingénierie, les modèles ont toujours tenu une place importante et ce depuis l'antiquité. [117] fait remarquer que Vitruve [1], un ingénieur romain du premier siècle avant JC discutait déjà de l'efficacité des modèles dans son manuel d'ingénierie [2]. Aujourd'hui en informatique, la discipline principalement associée au processus de modélisation est appelée l'Ingénierie Dirigée par les Modèles (IDM). Elle regroupe l'ensemble des techniques permettant de représenter des éléments et des concepts de la réalité sous forme de modèles. Mais elle regroupe également les méthodes et outils permettant de raisonner sur ces modèles.

2.1.1 Notion de modèle

Qu'est-ce qu'un modèle ?

La notion de modèle est centrale en IDM. Sa définition varie selon les membres de la communauté informatique. Considérons la définition donnée par [23] : *"A model is a simplification of a system built with an intended goal in mind [...]. The model should be able to answer questions in place of the actual system"* [3]. Cette définition attire l'attention sur le but du modèle et sur sa capacité à se substituer au système réel pour pouvoir raisonner. La définition donnée par [114] : *"A model is a representation in a certain medium of something in the same or other medium. The model captures the*

1. De son nom complet : Marcus Vitruvius Pollio.
2. The ten books on architecture.
3. En français : "un modèle est une simplification d'un système construit avec un but précis en tête [...]. Le modèle doit être capable de répondre à des questions à la place du système réel".

important aspects of the thing being modeled from a certain point of view and simplifies or omits the rest."[4], insiste sur la nature du modèle et notamment sur la mise en valeur des aspects souhaités du système réel et sur la simplification du reste.

Dans cette thèse, nous considérons la définition suivante d'un modèle qui unit les deux précédentes : *"Un modèle est une représentation partielle et suffisante d'un système réel suivant un certain point de vue. Il permet de s'abstraire de ce système pour répondre à une problématique donnée".*

Au delà de cette définition, un modèle s'exprime à l'aide d'un langage de modélisation. Ce langage est composé d'une syntaxe et d'une sémantique. La syntaxe correspond à la représentation des différents concepts et éléments constituant le langage. Ces éléments sont manipulables et visibles par l'utilisateur. En complément, la sémantique constitue le sens donné à chacun de ces éléments ou concepts. [114] décrit un modèle comme étant directement constitué d'une sémantique et d'une notation. En comparaison, [26] décompose un langage de modélisation entre une syntaxe concrète, qui est la syntaxe manipulée par l'utilisateur, et une syntaxe abstraite, qui est la syntaxe manipulée par la machine et épurée du sucre syntaxique profitable à l'utilisateur. De plus, il considère qu'il existe un domaine sémantique qui représente l'ensemble des états atteignables du système (i.e. l'ensemble des concepts exprimables dans le langage) et que la sémantique du langage correspond au lien entre les éléments de la syntaxe abstraite et ceux du domaine sémantique. Les travaux de [71] décrivent un langage de modélisation comme étant constitué d'une syntaxe, d'un domaine sémantique et des liens sémantiques entre cette syntaxe et ce domaine. Dans le cadre de cette thèse, nous considérons la décomposition basique d'un langage de modélisation en une syntaxe et une sémantique. Cette définition est suffisante pour la description de nos travaux.

Modèle contemplatif vs. Modèle productif

Bien que le principe de modélisation existe depuis de nombreuses années en informatique, les premières approches le mettant en pratique étaient principalement dédiées à des activités humaines d'ingénierie, comme par exemple l'approche Merise [123]. Ces approches se basent principalement sur des modèles de type contemplatif. La définition d'un modèle contemplatif est donnée par [18] : un modèle contemplatif est un modèle interprétable et manipulable par un être humain.

L'utilisation de modèles contemplatifs a eu un impact limité sur la production logicielle [18]. L'émergence de l'IDM offre une nouvelle manière de considérer le modèle dans le cadre d'un développement logiciel, le but n'étant plus de voir le code comme l'élément central de production mais de s'en abstraire au profit des modèles tout en restant productif [17]. Il n'est plus alors question d'un modèle contemplatif mais d'un modèle productif. La définition d'un modèle productif est donnée par [18] : un modèle productif est un modèle interprétable et manipulable par une machine.

4. En français : "un modèle est une représentation de quelque chose exprimée dans un certain moyen d'expression dans le même ou dans un autre moyen d'expression. Le modèle décrit les aspects importants de l'objet modélisé selon un certain point de vue et simplifie, ou omet, le reste".

2.1.2 Notion de méta-modèle

La logique de méta-modélisation

Afin qu'un modèle soit interprétable et manipulable par une machine, il faut que son langage de modélisation soit formellement défini. Dans une approche telle que l'IDM, la définition de ce langage est elle-même naturellement réalisée à l'aide d'un modèle. Ce modèle est appelé un méta-modèle. Cette notion est définie dans [95] : *"A model that defines the language for expressing a model"*[5]. Il ne faut cependant pas supposer que le méta-modèle représente la sémantique du langage. [71] rappelle que c'est une erreur commune et qu'un méta-modèle décrit seulement la syntaxe du langage.

S'il est possible de voir le méta-modèle comme un modèle, alors ce modèle peut posséder lui aussi un méta-modèle. Ce méta-modèle du méta-modèle se nomme alors le méta-méta-modèle.

Lors de l'émergence de la notion de modélisation et du "tout est modèle", il était nécessaire d'établir une base théorique pour la méta-modélisation afin de fédérer la création de méta-modèles [17] et ainsi éviter la prolifération de méta-modèles incompatibles. Afin de limiter les niveaux d'abstraction possibles, la structure hiérarchique d'une modélisation s'organise historiquement autour d'une architecture sur 4 niveaux. Ces 4 niveaux d'abstraction (ou de modélisation) sont :

- *1er niveau (M0) : le système* : le système tel qu'il existe dans le monde réel. Il est représenté par le modèle défini au niveau M1.
- *2e niveau (M1) : le modèle* : la représentation du système dans le langage de modélisation donné. Il doit être conforme au méta-modèle défini en M2.
- *3e niveau (M2) : le méta-modèle* : la définition du langage de modélisation. Il doit être conforme au méta-méta-modèle défini en M3.
- *4e niveau (M3) : le méta-méta-modèle* : il définit l'architecture du méta-modèle. Il a la propriété de se définir lui-même : c'est l'élément avec le plus haut niveau d'abstraction. En conséquence, il doit être conforme à lui-même.

La représentation la plus classique de cette architecture est donnée dans la Figure 2.1.

Cette approche hiérarchique est largement adoptée dans le cadre de l'IDM. Elle peut être représentée sous différentes formes [45]. Par exemple, [8] décrit cette architecture en y ajoutant des dimensions ontologiques et linguistiques à chaque niveau. D'une autre manière, [17] décrit cette architecture comme étant en fait une architecture de type 3+1 : il donne une dimension supplémentaire à cette architecture en regroupant les trois derniers niveaux (M1 à M3) sous la forme d'un ensemble nommé "le monde de la modélisation" et le dernier niveau (M0) sous la forme de l'ensemble nommé "le monde réel". Mais quelles que soient les versions existantes, l'architecture hiérarchisée par niveau de méta-modélisation est toujours présente.

Cette structure hiérarchique n'est pas propre à l'IDM puisqu'elle est utilisée depuis longtemps dans d'autres domaines informatiques. Par exemple, [46] explique que ce type de hiérarchie est propre à n'importe quel espace technique. Les auteurs définissent un

5. En français : "Un méta-modèle est un modèle qui définit le langage de modélisation d'un modèle".

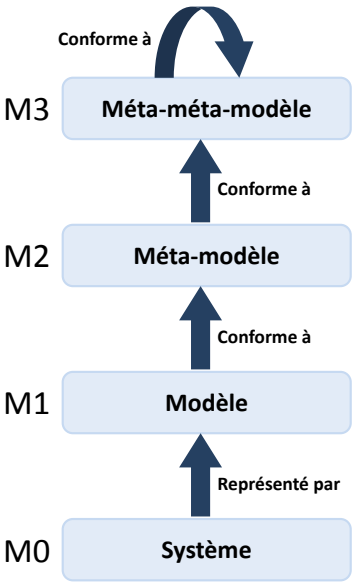


FIGURE 2.1 – Architecture sur 4 niveaux

espace technique comme *"un ensemble d'outils et techniques issus d'une pyramide de méta-modèles dont le sommet est occupé par une famille de méta-méta-modèles similaires"*. Ils considèrent alors l'espace technique des modèles [6], structuré comme présenté Figure 2.1. Ils donnent ensuite pour exemple l'espace technique des grammaires [7] comme présenté Figure 2.2. Dans cet espace technique, le programme d'un logiciel est situé au niveau M1 et il est l'équivalent d'un modèle en IDM. Il est exprimé à partir d'une grammaire de langage de programmation située au niveau M2, cette grammaire étant équivalente à un méta-modèle en IDM. Le niveau M3 contient un langage de description de grammaires, équivalent à un méta-méta-modèle en IDM.

Le Model Driven Architecture (MDA), prémisse à l'IDM

Bien que cette structure hiérarchique ne soit pas propre à l'IDM, l'IDM a toutefois été marquée par une approche basée sur ce type de structures. En 2000, [120] explique que la prolifération des middlewares [8] dans l'industrie devient problématique. Il était devenu difficile pour une entreprise de se reposer sur une seule plate-forme middleware. Et même lorsqu'une entreprise était capable de s'appuyer sur une unique plate-forme en interne, le besoin d'interopérabilité avec des entreprises partenaires imposait le plus souvent de travailler avec une technologie différente. Afin de pouvoir maîtriser et gérer cette multiplicité des plate-formes middleware, et plus précisément ce besoin d'interopérabilité,

6. En anglais : modelware.
7. En anglais : grammarware.
8. En français : intergiciel. Un intergiciel est un logiciel servant d'intermédiaire entre différentes applications.

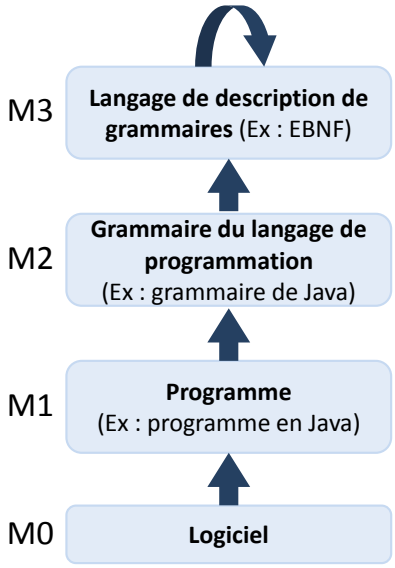


FIGURE 2.2 – Architecture de l'espace technique grammarware

l'Object Management Group (OMG) [9] a proposé une approche appliquant les techniques de modélisation : le Model Driven Architecture [120] (MDA).

L'objectif du MDA est de pouvoir permettre la création de modèles productifs d'applications, pérennes, stables et indépendants des plate-formes. Il est alors possible de tirer profit de ces modèles pour obtenir du code pour des plate-formes ciblées, plate-formes qui évoluent continuellement.

Le cœur de l'approche MDA se base sur un ensemble de standards créés par l'OMG. Ainsi, son architecture sur quatre niveaux se base sur un méta-méta-modèle nommé Meta Object Facility (MOF) [95], qui est un standard pour la méta-modélisation. L'approche s'appuie également sur le langage de modélisation Unified Model Language (UML) [98], dont le méta-modèle est basé sur le MOF, afin de pouvoir exprimer des modèles d'application indépendants des plate-formes [10] tels que des modèles de conception, et des modèles sur les parties dépendantes des plate-formes [11], c'est à dire des modèles liés au code.

La version finale de cette approche a été publiée en 2003 [96]. Elle est toujours d'actualité et il existe de nombreux retours de réussite de son application dans l'industrie [12].

9. L'OMG est un consortium international à but non lucratif spécialisé dans la définition de standards pour l'industrie.
10. En anglais : Platform Independant Model ou PIM.
11. En anglais : Platform Specific Model ou PSM.
12. `http://www.omg.org/mda/products_success.htm`

2.1.3 Pourquoi modéliser ?

D'un point de vue de l'utilisateur, le besoin de modéliser peut se manifester pour plusieurs raisons. Nous en distinguerons trois.

La maîtrise de la complexité

L'accroissement de la complexité des systèmes au cours des dernières années est une réalité. Par exemple, dans le domaine de l'aéronautique, [107] explique que l'avionique des Airbus A310 en 1980 représentait 4 Mo de codes et s'appuyait sur 77 calculateurs et 136 bus numériques. En comparaison en 1990, sur l'A340, l'avionique représentait 20 Mo de code et s'appuyait sur 115 calculateurs et 368 bus numériques. Pour l'avionique de l'A380 en 2005, [62] prévoyait que le code représenterait plusieurs centaines de Mo. Nous pouvons donc observer un accroissement approximatif de 2500% de la taille de leur code avionique au cours de ces 30 dernières années. Afin de pouvoir maîtriser cette complexité croissante, il est nécessaire de s'en abstraire et de représenter le système sous une forme différente et adaptée à la problématique à laquelle l'utilisateur souhaite répondre sur le système.

Le besoin de formaliser

La nécessité de formaliser est également une motivation pour utiliser un modèle. Le langage naturel peut se montrer source d'ambiguïtés. La manière d'interpréter une phrase varie selon la personne qui la lit. Dans le cadre du développement logiciel, [89] rapporte que des spécifications exprimées en langage naturel sont souvent source d'ambiguïtés et qu'il est nécessaire d'exprimer formellement ces spécifications. De plus, dans le contexte d'entreprise étendue décrit précédemment qui entoure les projets actuels, il est indispensable de communiquer dans un langage formel et commun pour partager l'information. L'utilisation des modèles dans ce cadre est alors un moyen de formalisation.

La transformation de modèles : intéropérabilité et raffinement

Il est aussi possible de modéliser dans le but de transformer. Dans la littérature [32, 64], deux principaux types de transformation sont caractérisés :

- la transformation d'un modèle vers un modèle [13]. Elle va particulièrement être efficace pour passer d'un formalisme vers un autre. Par exemple, [26] explique que ce type de transformation peut s'utiliser dans le cadre d'une migration logiciel d'une plate-forme à une autre. Elle permet également de raffiner un modèle abstrait vers un modèle plus spécifique. Par exemple, [10] explique qu'il est possible de transformer un modèle abstrait vers un modèle d'implémentation plus détaillé. Citons également la possibilité de passer d'un modèle indépendant des plate-formes (PIM) vers un modèle dépendant d'une plate-forme (PSM) comme le propose l'approche MDA décrite précédemment.

13. En anglais : model-to-model.

– la transformation d'un modèle vers du texte [14]. En premier lieu, elle permet de générer du code. Un même modèle pourra ainsi permettre de générer du code dans différents langages suivant des transformations. Elle permet également de générer des tests à partir d'un modèle [73]. Finalement, ce type de transformation permet la génération de documents. Il est en effet nécessaire dans certains développements logiciels d'obtenir de la documentation. Par exemple dans [51], nous générons de la documentation à partir d'un modèle à l'aide de l'outil GenDoc2 [15] basé sur la technologie Eclipse Model-To-Text [16] afin de pouvoir faire de la traçabilité sur des exigences. Dans [108], les auteurs utilisent ce même outil pour générer de la documentation à partir de modèles en vue de répondre à un standard pour le développement logiciel.

2.1.4 De nombreux langages pour modéliser

La popularité de l'approche IDM a engendré la prolifération des langages de modélisation, chacun proposant de s'adapter à des besoins ou à des domaines spécifiques.

UML

Le standard UML [17] est un langage de modélisation défini par l'Object Management Group. C'est un standard très largement utilisé dans l'industrie du logiciel. Il se base sur le méta-méta-modèle MOF dont nous avons parlé précédemment et il représente le langage de modélisation dont dépend l'approche MDA.

Le but d'UML est de pouvoir représenter le comportement, l'architecture, les structures de données et la structure elle-même d'une application. Ces différentes représentations s'appuient sur différents diagrammes, chacun proposant de représenter le système suivant un point de vue particulier. UML propose treize diagrammes différents répartis en trois catégories. La première catégorie regroupe les diagrammes structurels avec le diagramme de classe, le diagramme d'objets, le diagramme de composant, le diagramme de structure composite, le diagramme de package et le diagramme de déploiement ; la seconde catégorie concerne les diagrammes comportementaux avec le diagramme d'activité, le diagramme de machine à états et le diagramme de cas d'utilisation ; la dernière catégorie rassemble les diagrammes d'interaction avec le diagramme de séquence, le diagramme de communication, le diagramme de temps et le diagramme de collaboration.

Bien entendu, il n'est pas imposé d'utiliser tous ces diagrammes au cours d'un développement logiciel mais seulement ceux nécessaires au développement de l'application. Le méta-modèle de chacun de ces types de modèles est décrit dans [98], la sémantique y est décrite en langage naturel et dans un langage de contraintes nommé OCL [99], lui-même standard défini par l'OMG et s'appliquant au standard UML.

14. En anglais : model-to-text.
15. `http://marketplace.eclipse.org/content/gendoc2`
16. `http://www.eclipse.org/modeling/m2t/`
17. `www.uml.org/`

Une autre capacité d'UML est la possibilité d'étendre formellement le langage grâce à son mécanisme d'extension par profil. Ainsi, l'utilisateur est libre de créer sa propre extension du langage correspondant à ses besoins. Par exemple, le profil UML MARTE [18] est un standard spécifié par l'OMG pour la modélisation des systèmes embarqués temps réel [68]. Il spécifie des concepts pour caractériser des éléments UML permettant de représenter les notions de temps, de concurrence, de plate-forme logicielle et matérielle, de ressources et de caractéristiques quantitatives sur un système, telles que le temps d'exécution. Il est organisé hiérarchiquement sous forme de packages regroupant chacun un aspect particulier du développement de logiciel embarqué temps réel. Nous distinguons principalement des packages dédiés à la conception du système et d'autres dédiés à l'analyse et à la vérification. Les spécificités pour la conception regroupent la modélisation des mécanismes temporels, des ressources, des composants et de leurs moyens de communication. Les packages dédiés à l'analyse et à la vérification permettent d'annoter les modèles pour conduire des analyses d'ordonnançabilité ou de performances. Le profil fournit également la possibilité de spécifier des propriétés non-fonctionnelles sur un système.

SysML

Le standard SysML [19] [100] est un langage de modélisation défini par l'OMG. Il s'agit d'un profil standardisé UML se présentant sous la forme d'une extension d'un sous-ensemble d'UML. Face au succès d'UML, le but de SysML n'est plus de se cantonner au monde de l'informatique mais de proposer un langage de modélisation utilisable pour décrire tout système, qu'il soit logiciel ou non. Le standard SysML propose notamment des activités de spécification, d'analyse, de conception et de vérification de systèmes et de systèmes de systèmes.

Le langage se base sur certains diagrammes UML et en ajoute également de nouveaux. Le diagramme de blocs remplace le diagramme de classe et permet ainsi non plus de représenter des classes logicielles, mais des éléments comme du matériel, du logiciel ou d'autres éléments d'un système. Un diagramme d'exigence est également ajouté par rapport à UML permettant de faire le lien entre les éléments modélisés et les exigences. Le diagramme paramétrique va permettre de représenter des contraintes sur des éléments du système telles que des contraintes de performance ou de fiabilité. Ce type de diagramme va être particulièrement utile dans le cadre de l'analyse de modèles.

Basé sur UML, SysML offre également un mécanisme d'extension par profil. Le profil SysML AVATAR [102] est un exemple de spécialisation au domaine embarqué qui offre des solutions pour la modélisation et la vérification formelle de systèmes embarqués temps réel. Il permet de supporter des activités réalisées en amont d'un cycle de développement, en se basant sur une partie des diagrammes du langage SysML et de stéréotypes appliqués à leurs éléments. Il propose notamment des solutions pour le recueil des exigences, l'analyse du système, la modélisation ainsi que l'expression de propriétés

18. Modeling and Analysis of Real-Time and Embedded Systems.
19. Systems Modeling Language, `http://www.omgsysml.org/`

de sûreté et de sécurité.

Le langage Simulink

Ce que nous appelons le langage Simulink est le langage tiré de l'environnement de modélisation Simulink [20] mis au point par la société MathWorks. Ce langage graphique permet de représenter le modèle fonctionnel d'un système à l'aide de blocs et de signaux. Les blocs sont reliés par des signaux et chaque bloc représente un traitement particulier des signaux. Une bibliothèque de blocs est disponible dans l'environnement et va fournir des blocs pour des traitements dynamiques, algorithmiques ou structurels des données. Les signaux du modèle vont pouvoir transporter une donnée ou un événement.

L'utilisation de modèles Simulink est principalement dédiée à la modélisation de systèmes dynamiques à temps discret ou continu tels que les systèmes de communication et de traitement du signal.

D'autres langages de modélisation

De nombreux autres langages de modélisation existent mais il serait vain de chercher ici à en donner une liste exhaustive. Nous pouvons toutefois citer quelques autres standards. Du point de vue de l'architecture, le langage AADL [21] [47] est un standard défini par SAE [22] pour la modélisation et l'analyse d'architectures système. Il est principalement utilisé pour la modélisation de systèmes embarqués. Pour la représentation de processus, le standard BPMN [23] [97] est un langage de modélisation pour la représentation de processus métier défini par l'OMG. Citons également à titre d'exemple, le langage SDL [24] qui permet la modélisation de systèmes de télécommunication. Il est défini comme une norme par l'ITU [25].

Les langages dédiés ou DSL

Même si des standards ont été définis, il est parfois nécessaire de définir un langage dédié à son domaine ou à ses besoins. Nous avons vu que le standard UML possédait un système d'extension pour répondre à ce genre de besoin. En IDM, il également possible pour chaque utilisateur de définir son propre langage de modélisation dédié. Ces langages dédiés, ou plus simplement DSL [26], sont en général plus petits, plus simples à appréhender et plus simples à manipuler que des standards génériques tels que UML. Ils sont egalement plus expressifs de part leur spécialisation à un domaine. De plus, l'utilisation d'un méta-méta-modèle standard tel que le MOF dans le cadre de la définition de ces langages de modélisation va également permettre la définition de transformations entre

20. `http://www.mathworks.fr/products/simulink`
21. Architecture Analysis & Design Language.
22. Society of Automotive Engineers.
23. Business Process Model and Notation.
24. Specification and Description Language.
25. International Telecommunication Union.
26. Domain Specific Language.

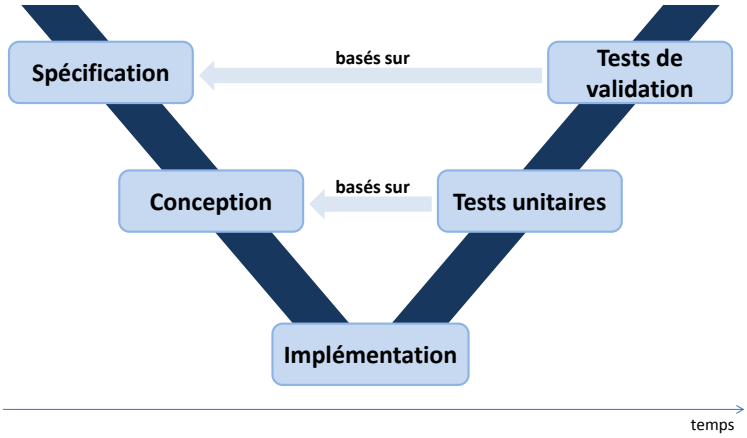


FIGURE 2.3 – Processus de développement logiciel en V

modèles issus de différents DSLs et ainsi multiplier les possibilités d'usage de ces DSLs. La définition et l'utilisation de ces DSLs peuvent alors être vues comme l'application la plus directe de l'approche IDM.

Il existe des langages et outils permettant la création de DSL. Par exemple, le langage de méta-modélisation Ecore disponible à travers l'outil Eclipse EMF [27] [122] permet la définition de DSLs. Cet outillage offre alors des alternatives pour gérer et manipuler ces DSLs basés sur Ecore.

2.1.5 L'intégration de l'IDM dans les processus de développement

Industriellement, le développement d'un logiciel se base principalement sur trois contraintes : le délai de production, le coût de production et la qualité de la production. L'enjeu du développement est donc de produire un système suffisamment bon selon un contexte d'utilisation en respectant des coûts et des temps de production raisonnables [46], sachant qu'il faudrait sans doute un temps infini pour construire un système parfait. Le génie logiciel définit les bonnes pratiques ou "règles de l'art" de l'ingénierie pour produire des systèmes logiciels. Dans ce cadre, il précise les différentes étapes qu'un processus de développement doit suivre pour produire un logiciel.

Il existe différents processus de développement logiciel. Les différentes étapes du développement y sont similaires mais le nombre d'itérations sur certaines étapes ou les liens entre elles sont susceptibles de varier. Par exemple, la Figure 2.3 représente le processus de développement logiciel le plus classique basé sur le modèle dit en "V". Il est composé des phases les plus représentatives d'un développement logiciel : la spécification

27. Eclipse Modeling Framework.

des exigences, la conception du logiciel, l'implémentation du logiciel et les phases de tests (unitaires ou de validation).

Nous avons vu dans 2.1.3 que l'IDM propose un certain nombre d'activités pour répondre à des besoins actuels en terme de développement logiciel. Chacune de ses activités peut s'intégrer dans des étapes d'un processus de développement. Nous donnons dans les paragraphes suivants un aperçu des possibilités pour chaque étape du processus ainsi qu'un échantillon de retours industriels sur les gains en productivité et en qualité apportés par l'usage de l'IDM au cours d'un développement logiciel.

L'IDM et les phases de spécification et de conception

Les phases de spécification et de conception permettent la définition et la prise en compte des exigences pour la création du produit attendu. Dans ce contexte, les besoins de maîtriser la complexité, communiquer ou formaliser ses idées peuvent être satisfaits grâce à l'utilisation des modèles. Cependant, l'utilisation des modèles va également offrir d'autres possibilités pour ces phases du processus de développement. Comme l'explique [46], chaque phase du processus de développement implique la création d'artefacts, ou ce que nous appelons des livrables. Par exemple, dans le cadre de l'aéronautique, la norme DO-178C impose pour chaque phase du processus, un certain nombre de documents à fournir aux autorités compétentes dans le cadre de la certification du logiciel. Grâce aux transformations de modèles, l'IDM va pouvoir permettre la génération automatique de certains de ces livrables comme par exemple la génération de la documentation comme décrit dans la Section 2.1.3.

La possibilité de travailler avec des modèles formels au niveau de la spécification et de la conception va également permettre de réaliser et d'automatiser une autre tâche importante : la vérification. Cette activité est d'autant plus essentielle que [117] explique qu'il est crucial de prévoir des vérifications au plus tôt du cycle de développement étant donné que les décisions fondamentales concernant la conception du logiciel y sont prises.

Dans la littérature, de nombreux travaux existent sur la vérification de modèles. Par exemple, si nous prenons uniquement les modèles basés sur le standard UML, [119, 5, 24] proposent des approches pour la vérification de modèles UML par rapport à des propriétés exprimées sous forme de contraintes en OCL. L'approche qui est présentée dans [5] s'appuie sur l'utilisation de l'outil UML2Alloy, qui implémente une transformation de la spécification en UML et des contraintes OCL vers une spécification en Alloy [28], afin de procéder à leur vérification à l'aide du solveur Alloy Analyzer. Le langage Alloy est un langage de modélisation pour la conception logicielle, basé sur la logique du premier ordre. En comparaison, la méthode présentée dans [24] se base sur la transformation d'un diagramme de classes UML et de contraintes OCL en un problème de Satisfaction de Contraintes (CSP) [29]. Le problème CSP peut ensuite être résolu à l'aide d'un solveur.

28. `http://alloy.mit.edu/alloy/`

29. Un problème de satisfaction de contraintes est un problème composé d'un ensemble fini de variables, d'un ensemble de domaines de valeurs (un domaine pour chaque variable) et un ensemble de contraintes portant sur les variables. Une solution à un CSP, est l'assignation de valeurs aux variables en satisfaisant toutes les contraintes, valeurs comprises dans le domaine de valeur de chaque variable respective.

Ces travaux présentent également l'implémentation d'un prototype, nommé UMLtoCSP, automatisant la transformation et le lien vers le solveur ECLiPSe [30]. La vérification de modèle UML présentée dans [119] s'appuie sur la transformation d'un diagramme de classe UML et de contraintes OCL vers un problème SAT [31]. Le problème SAT est ensuite résolu via le solveur MiniSat [32].

Enfin des travaux plus récents ne se limitent plus à vérifier un modèle lors des phases de spécification ou de conception, mais tendent à générer des modèles en fonction d'un ensemble d'exigences d'entrée. Par exemple, [33, 34, 36] proposent des approches permettant à partir d'un modèle partiel et d'un ensemble de contraintes de générer des modèles complets et optimaux par rapport aux contraintes d'entrée. Similairement aux approches décrites précédemment, ces travaux se basent sur des techniques de transformation de modèles vers des problèmes de résolutions de contraintes.

L'IDM et la phase d'implémentation

L'utilisation des modèles dans les phases amont va permettre l'utilisation d'une approche de génération de code pour la phase d'implémentation. Il va être possible, à partir du modèle, de générer partiellement ou complètement le code du logiciel attendu. Cette génération de code offre plusieurs avantages. Selon [46], cette technique permet de rester flexible par rapport à l'évolution des langages de modélisation : si le langage de modélisation est modifié, il suffit de rectifier la transformation de modèles pour générer le code. Cette technique permet également de rester flexible face à la diversité des plate-formes et systèmes ciblés : comme vu en 2.1.3, un même modèle pourra servir à générer du code dans différents langages suivant la transformation de modèle réalisée.

Parmi les générateurs de code existants, citons le générateur GeneAuto [125] qui permet de générer automatiquement du code C à partir d'un modèle Simulink. Il est issu d'un projet européen basé sur une collaboration à la fois d'acteurs académiques tels que l'IRIT et INRIA et d'industriels tels que Airbus, Thales et Continental. Citons également les générateurs présents dans l'outil SCADE Suite [33] qui permettent de générer du code C ou Ada à partir d'un modèle Scade. Le générateur de code C est notamment qualifié : le code généré n'a plus besoin d'être testé, le générateur garantit qu'il correspond au modèle d'entrée, qui, lui, est vérifié. Plus récemment, le projet P [34] qui est un projet de recherche financé par le Fond Unique Interministériel 2011, a pour but la création d'un générateur open-source de code qualifié à partir de modèles pour les systèmes embarqués temps-réel. Il a pour objectif de prendre en entrée des modèles de type UML, SysML, Simulink et de générer du code C/C++ ou Ada.

30. `http://eclipseclp.org/`
31. Abréviation de Boolean Satisfiability Problem, ou Problème de satisfiabilité. Un problème SAT est un problème visant à déterminer si pour une formule composée de variables booléennes, il existe des valeurs pouvant être assignées à ces variables et permettant d'évaluer la formule à vraie. S'il existe de telles valeurs, le problème est satisfiable. Sinon, le problème est dit insatisfiable.
32. `http://minisat.se/`
33. `http://www.esterel-technologies.com/products/scade-suite/`
34. `http://www.open-do.org/projects/p/`

L'IDM et la phase de tests

La phase de tests permet de vérifier que le système produit possède bien le comportement attendu. Comme [46] l'explique, ce comportement est le plus souvent décrit dans un document de spécification et le système à tester correspond le plus souvent à une implémentation. Dans le cadre d'une approche IDM, le document de spécification est exprimé à partir de modèles. Il est alors possible de générer automatiquement des tests à partir du modèle de spécification.

[46] fait d'ailleurs référence à la technique de *model-based testing* qui utilise un modèle de spécification décrivant le comportement attendu de l'implémentation du système pour générer les tests. Ces tests se composent de cas de test, chacun représentant un chemin d'exécution possible parmi l'ensemble des chemins décrits dans le modèle de spécification. La génération de tests consiste alors à générer un ensemble de cas de test répondant à un objectif de couverture du modèle de spécification. [128] explique que ces cas de test sont tout d'abord générés sous forme de cas de test *"abstraits"* à partir du modèle. Un cas de test *"abstrait"* est un cas de test qui n'est pas directement exécutable. Par exemple, un cas de test abstrait pourrait être exprimé dans un diagramme de séquence UML. A partir de ces cas de test abstraits, il alors possible de générer des cas de test exécutables sur l'implémentation du système ciblé. [128] donne notamment des exemples de ce type de génération à partir de diagrammes UML pour des implémentations en Java.

Les retours industriels

Théoriquement le passage à une approche IDM dans le cadre d'un processus de développement est censé apporter des gains, aussi bien au niveau de la productivité que de la qualité du système produit. Cependant, il est justifié de se demander si l'utilisation industrielle de cette approche tient réellement ses promesses. [90] détaille une étude réalisée sur une vingtaine articles portant sur les retours d'expériences de l'usage de l'IDM sur des cas d'études industriels et publiés entre 2000 et 2007. Nous allons ici revenir sur quelques résultats de cette étude.

[90] expose tout d'abord les gains en productivité apportés par l'IDM à partir des travaux réalisés par Motorola [9, 130] dans un contexte de développement de système de télécommunication. L'IDM y est appliquée de la phase de spécification jusqu'à la génération de code et de tests. Par exemple, dans [9], les auteurs estiment que la productivité a été multipliée entre 2 et 8 fois pour la production de lignes de code par rapport à une approche classique de codage manuel. Dans [130], les auteurs expliquent que pour un cycle de tests comprenant une étape de correction du modèle, une étape de création d'un cas de tests, une étape de génération du code corrigé et une étape de tests de non-régression, la durée de ce cycle, mesurée pour quatre versions de différentes fonctions réseaux, est passée d'environ 25-75 jours à 24 heures. Dans un autre exemple [124], l'entreprise The Middleware Company a réalisé une étude comparative sur 2 équipes développant la même application, l'une utilisant l'IDM au travers du MDA et l'autre non. L'étude a montré que l'équipe utilisant l'IDM a pu développer l'application 35% plus vite que l'autre équipe.

Concernant le gain de qualité, [90] explique que peu de données quantitatives existent dans la littérature. Cependant, il présente l'article [9] où les auteurs exposent qu'ils ont pu observer au cours de leurs expérimentations une réduction de 1,2 à 4 fois des erreurs lors de l'usage de l'IDM. Ils font également remarquer que la plupart des erreurs sont trouvées beaucoup plus tôt dans le processus de développement, engendrant ainsi une baisse des coûts de correction.

Le nombre restreint de retours d'expériences disponibles, de données quantitatives ou tout simplement de résultats portant sur des échecs de la mise en place de l'IDM font que les résultats de l'étude de [90] ne permettent pas de donner un bilan généralisable à l'ensemble de l'approche mais ils permettent néanmoins de donner un aperçu des possibilités et des gains apportés par l'IDM.

2.1.6 Des outils pour appliquer l'IDM

Afin de proposer à l'utilisateur un support outillé aux activités liées à l'IDM, des plate-formes intégrant ces activités et leurs automatisations ont vu le jour. En voici un aperçu.

Eclipse Modeling Project

L'Eclipse Modeling Project [35] [67] est un projet Eclipse qui a pour but de favoriser des activités IDM dans Eclipse. Plus particulièrement, il regroupe un ensemble d'outils unifiés pour la modélisation de DSL et l'implémentation de standards pour la communauté Eclipse.

Ce projet s'organise en plusieurs sous-projets, chacun supportant une activité particulière de l'IDM pour l'utilisateur. Tout d'abord, il propose des outils pour le développement de syntaxes abstraites de DSL. Le but étant de permettre l'édition, la vérification et l'interrogation de modèles. Ces activités sont supportées par le sous-projet Eclipse EMF. L'Eclipse Modeling Project propose ensuite un outillage pour le développement de syntaxes concrètes de DSL. Cet outillage doit permettre la création d'une syntaxe concrète, textuelle ou graphique, à partir d'une syntaxe abstraite. C'est le sous-projet Eclipse GMP [36] qui permet la création d'éditeurs graphiques et le sous-projet Eclipse TMF [37] qui permet la création d'éditeurs textuels. L'Eclipse Modeling Project propose également un ensemble d'outils pour la transformation de modèles au travers des outillages Eclipse MMT [38] et Eclipse M2T [39]. Enfin, un ensemble de standards de modélisation, tels que par exemple UML et BPMN, sont implémentés dans Eclipse MDT [40]. Ces implémentations sont compatibles avec les technologies Eclipse, permettant l'utilisation des différents outils IDM existants sur la plate-forme.

35. `www.eclipse.org/modeling`
36. Graphical Modeling Project, `www.eclipse.org/modeling/gmp`
37. Textual Modeling Framework, `www.eclipse.org/modeling/tmf`
38. Model to Model Transformation, `www.eclipse.org/mmt`
39. Model To Text, `www.eclipse.org/modeling/m2t`
40. Modeling Development Tools, `www.eclipse.org/modeling/mdt`

L'outil Eclipse est très répandu dans le milieu industriel ce qui permet au projet Eclipse Modeling Project de profiter d'une très large communauté.

TOPCASED

Topcased [41] [43] (Toolkit in Open Source for Critical Applications & Systems Development) est un atelier logiciel open-source destiné à supporter l'activité d'Ingénierie Dirigée par les Modèles pour le développement des systèmes critiques. Il a été initié en 2004 par le CNRT-AE [42] et a été soutenu par plus d'une trentaine de partenaires à la fois académiques et industriels dont Airbus, Thales, Atos et l'ONERA.

Il se présente sous la forme d'une Plate-forme Client Riche (RCP) Eclipse et possède ainsi une architecture modulaire qui lui permet de regrouper un grand nombre d'outils. Il se base notamment sur les technologies issues de l'Eclipse Modeling Project. Cette architecture favorise également la création et l'ajout de fonctionnalités ou d'outils personnalisés sous la forme de greffon (plug-in) Eclipse. D'autre part, la technologie Eclipse, basée sur le langage Java, lui assure également une vaste portabilité. Le principal avantage de Topcased est de fournir, dans un même environnement et pour la communauté des systèmes critiques, des outils pour la modélisation, la traçabilité des exigences, la simulation de modèles, la vérification de modèles, la transformation de modèles, l'implémentation, la génération de code, la rétro-ingénierie, le travail collaboratif et la génération documentaire. Il supporte notamment les standards UML, SysML et AADL mais permet également la définition de DSL grâce au langage Ecore issu de la technologie Eclipse.

Topcased est un outil vivant et largement plébiscité. Il enregistrait dernièrement un taux moyen de plus de 7000 téléchargements par mois pour la période allant de Septembre 2012 à Juin 2013 [43].

SCADE Suite

SCADE Suite [44] est un outil commercial développé par Esterel Technologies. C'est un environnement de développement basé modèle pour les logiciels critiques. Il se base sur le langage de modélisation graphique Scade qui permet la représentation de machines à états synchrones et de modèles de flots de données. Ce langage est lui-même basé sur un langage sous-jacent nommé Lustre [69], qui est un langage de programmation synchrone de flots de données.

SCADE Suite offre des activités basées modèle tout au long du processus de développement logiciel. Par exemple, durant la phase de conception, il permet à l'utilisateur de modéliser son système et de le vérifier, à l'aide de simulations ou de vérifications de propriétés de sûreté de fonctionnement. Durant la phase d'implémentation, il offre la

41. `www.topcased.org`
42. Centre National de Recherche Technologique Aéronautique et Espace.
43. Source : `www.topcased.org/index.php?indicators`
44. `www.esterel-technologies.com/products/scade-suite`

possibilité de générer du code à partir du modèle, en C ou en Ada. Ce code est qualifiable suivant les normes de certification telles que par exemple la norme DO-178B pour l'aéronautique.

SCADE Suite est également lié à d'autres produits SCADE comme SCADE Display [45] qui permet de créer rapidement des prototypes simples de systèmes de contrôle dans le but de simuler et vérifier le système et son interface. Scade Suite est largement utilisé dans l'industrie. Parmi les utilisateurs, il y a notamment : Airbus, EADS Astrium, Sagem, Eurocopter et l'U.S. Army.

Autres outils

Beaucoup d'autres outils proposent de supporter une activité d'Ingénierie Dirigée par les Modèles, de la conception jusqu'à la génération de code, malheureusement nous ne pouvons pas tous les citer. A titre d'exemple, l'environnement Simulink, dont nous avons décrit le langage dans 2.1.4, permet de créer des modèles fonctionnels de systèmes de traitement de signal et de communication, de simuler ces modèles, et d'en générer du code C. IBM a également mis au point son propre outillage pour le support de l'approche IDM. Les outils Rational tel que Rhapsody [46] permettent la modélisation, la vérification de modèles et la génération de code pour des applications logicielles. Ils prennent notamment en compte le standard UML. Rational Software Architecture [47] (RSA) permet lui la modélisation de processus d'entreprise. Enfin un autre exemple est l'outil MagicDraw [48] qui supporte également des activités IDM et qui se base sur le standard UML.

2.2 Machines à états et automates : représentation comportementale en informatique

Nous avons présenté dans la section précédente un état de l'art de l'Ingénierie Dirigée par les Modèles qui permet d'avoir un aperçu sur les possiblités de modélisation actuelles pour la représentation de système et/ou de logiciel et sur les avantages offerts par l'approche. Dans cette thèse, nous nous intéressons particulièrement aux techniques de représentation comportementale pour le logiciel. En ce sens, il est indispensable de s'intéresser aux notions d'automates, de machines à états et à leurs théories.

En informatique, le comportement d'un système est fréquemment représenté sous la forme d'une machine à états [40]. L'une des premières apparitions de la notion de machine à états dans l'informatique remonte aux années 50 lorsque Turing [127] cherchait à définir la notion de "machine pensante" et plus particulièrement de "calculateur numérique", en opposition au calculateur dit "humain". A l'époque, il fait l'analogie de ce type de

45. www.esterel-technologies.com/products/scade-display
46. www-03.ibm.com/software/products/fr/fr/ratirhapfami
47. www-03.ibm.com/software/products/fr/fr/ratisystarch
48. www.nomagic.com/products/magicdraw.html

		Etat courant		
		q_1	q_2	q_3
Entrée	i_0	q_2	q_3	q_1
	i_1	q_1	q_2	q_3

Etat	q_1	q_2	q_3
Sortie	o_0	o_0	o_1

FIGURE 2.4 – Exemple de machine à états discrets selon Turing

calculateur avec ce qu'il nomme une "machine à états discrets" [49] : *"These [Discrete state machines] are the machines which move by sudden jumps or clicks from one quite definite state to another. These states are sufficiently different for the possibility of confusion between them to be ignored. Strictly speaking there are no such machines. Everything really moves continuously. But there are many kinds of machines which can profitably be thought of as being discrete state machines."*[50]. Ces machines à états sont capables de calculer un nouvel état et une donnée de sortie en fonction de données d'entrée et de l'état courant. Elles sont composées d'un nombre fini d'états et peuvent être représentées sous la forme de deux tableaux : un tableau qui donne l'état de sortie suivant le signal d'entrée et l'état courant et un autre tableau qui donne le signal de sortie suivant l'état courant. Aucune représentation graphique n'est alors donnée à l'époque. Un exemple issu de [127] est représenté Figure 2.4. Ces deux tables représentent une machine à états discrets qui gère la rotation d'une roue au cours du temps. La roue peut être dans trois positions (q_1, q_2 ou q_3) et change de position chaque seconde. Ce changement peut être stoppé par l'activation d'un levier (i_1). Chaque position de la roue engendre l'éclairage d'une lampe (o_0 pour indiquer que la lampe n'est pas éclairée et o_1 pour indiquer que la lampe est éclairée).

Depuis, d'autres types de machines à états ont vu le jour. Cependant, il faut noter que la notion de machine à états est communément associée à la notion d'automate. Le flou qui règne entre les deux concepts se retrouvent très fréquemment dans le discours ou dans la littérature. A titre d'exemple, dans [94] l'auteur indique que les automates finis sont des machines à états. En comparaison, [110] sous-entend dans sa thèse que les machines à états font partie de la théorie des automates, dont est issu la notion d'automate en informatique. Dans cette thèse, nous considérons que les machines à états sont tout simplement un type d'automate.

2.2.1 La théorie des automates

Principes

Revenons très sommairement sur la théorie des automates afin de donner le contexte des origines du concept. La théorie des automates concerne l'étude des machines de calcul

49. En anglais : Discrete state machines.

50. En français : "Ce sont les machines qui évoluent par sauts ou clics instantanés d'un état prédéfini vers un autre. Ces états sont suffisamment différents pour éviter de les confondre. A proprement parler, il n'existe pas de telles machines. Tout est continuellement en mouvement. Cependant, il existe de nombreux types de machines qui peuvent être utilement représentés comme des machines à états discrets".

dites abstraites [77]. Ces machines de calcul abstraites sont représentées par des automates finis. Un automate fini est composé d'un ensemble d'états et de transitions entre ces états qui sont franchies en réponse à une donnée d'entrée. Plus formellement, [103] décompose un automate fini en un ensemble fini d'états, un alphabet [51] de symboles et un ensemble de transitions. Dans cet automate, une transition du type (s_1, a, s_2) spécifie que l'état s_1 est atteint à partir de l'état s_2 en lisant le symbole a. D'après [77], la théorie des automates englobe également les notions de "mot" et de "langage". Un mot est défini comme une séquence finie de symboles appartenant à un alphabet et un langage est défini comme une liste finie de mots. A partir de ces concepts, la théorie des automates définit un problème comme la question suivante : "Est ce que ce mot appartient à ce langage ?". Un automate fini est alors un moyen de résoudre ce problème.

Cette définition standard est assez abstraite mais permet de poser les bases de la théorie. D'un point de vue plus général, un automate fini va permettre, par exemple, de reconnaître des "mots", de vérifier qu'une suite d'états peut se produire ou non, ou tout simplement de vérifier si un état est atteignable ou non. [77] donne quelques exemples d'applications pratiques, telles que l'analyse lexicale d'un compilateur, la recherche d'occurrences de mots ou de phrases dans de vastes quantités de données ou bien encore la vérification des systèmes qui possèdent un nombre limité d'états distincts.

Historique

Les bases de la théorie sont issues de travaux réalisés principalement au début de la deuxième moitié du siècle dernier, bien que l'un des travaux pionnier de la théorie remonte à 1936 avec la machine de Turing [126]. [103] établit un historique de la théorie et explique que les fondements ont été introduits quelques années plus tard par des chercheurs qui sont issus de domaines parfois très différents. L'historique que nous allons présenter est un résumé des travaux de [103].

En 1948, les travaux de C.E. Shannon s'approchent du concept d'automate fini en s'intéressant à la définition d'une théorie mathématique pour la communication à l'aide de chaînes de Markoff [118]. C'est ensuite au tour de D.A. Huffman, en 1954, de s'approcher du concept d'automate dans ses travaux sur les circuits de commutation électroniques [79]. Il y définit notamment des diagrammes de transitions et ce qu'il appelle à l'époque des matrices de transitions. En 1956, S.C. Kleene publie un article [82] sur la représentation des événements dans les réseaux de neurones à partir d'automates finis.

Parmi les nombreux travaux de l'époque, citons deux types d'automates particuliers : les automates de Mealy [88] en 1955 et les automates de Moore [91] en 1956, également appelés machine de Mealy et machine de Moore. Les automates de Moore sont des automates composés d'un ensemble fini d'états, d'un ensemble fini de symboles d'entrée et d'un ensemble fini de symboles de sortie. Les transitions entre états dépendent du symbole d'entrée et c'est l'état courant qui détermine le symbole de sortie. Cet automate est donc assez proche du concept de machine à états discrets défini par Turing. Moore se sert de cet automate comme d'un modèle mathématique pour la résolution de problèmes.

51. Un alphabet est un ensemble fini et non-vide de symboles

En comparaison, Mealy s'inspire des travaux de Moore et de Huffman pour proposer un automate composé des mêmes éléments mais dont le symbole de sortie dépend à la fois de l'état courant et du symbole d'entrée. Il définit ces automates dans le cadre de ces travaux sur les circuits électroniques.

Les principales notions de la théorie des automates seront finalement présentées, selon [103], dans l'article [109] de M.O. Rabin et D. Scott en 1959. Depuis, de nombreux travaux ont suivi et traitent principalement de la définition de nouveaux types d'automates et de leurs applications. Nous pouvons notamment citer les automates de Büchi [22] qui sont l'un des types d'automate qui a marqué cette période. Dans ces travaux, J.R. Büchi étend la définition d'automate fini à la prise en compte de mots infinis. Dans ce cadre, il définit des états acceptants, c'est-à-dire des états dans lequel l'automate peut passer infiniment souvent. Un mot est alors reconnu comme valide si sa lecture par l'automate de Büchi passe infiniment par l'un des états dit "acceptant". Cette extension a notamment permis de s'intéresser aux comportements non-terminaux. Ces automates sont aujourd'hui utilisés dans le cadre du model-checking.

Bilan

Nous avons passé sous silence toute une partie de la théorie des automates, le but étant simplement de donner un aperçu suffisant des origines des automates afin d'introduire les différents types qu'il est possible de trouver actuellement. Néanmoins, il faut tout de même noter que la théorie des automates s'applique à différentes problématiques telles que la complexité des algorithmes, la décidabilité des problèmes, etc. Dans le cas où le lecteur souhaiterait avoir une description plus exhaustive de la théorie, nous l'invitons à lire l'ouvrage [77].

2.2.2 Différents types d'automates

Aujourd'hui, de nombreux types d'automates existent, chacun répondant à des besoins et à des problèmes différents. Dans notre contexte, nous nous intéressons exclusivement aux automates permettant la représentation comportementale de systèmes, le but de ces automates étant de donner une abstraction du système sur laquelle raisonner.

Les automates temporisés

Les automates temporisés [4] sont des automates permettant la représentation de systèmes temps-réel asynchrones. Comme l'explique [3], un système peut être modélisé comme un automate fini non-déterministe prenant en entrée des mots infinis, à la manière d'un automate de Büchi. Le langage reconnu par cet automate est alors l'ensemble des comportements du système.

Communément, un automate temporisé est composé d'un ensemble fini d'états (ou de localités), d'un alphabet d'actions et d'un ensemble fini d'horloge à valeurs dans $\mathbb{R}+$. Chaque horloge de l'automate démarre initialement à 0 et évolue de manière équivalente et continue. Les transitions entre états sont contraintes par une action de l'alphabet et

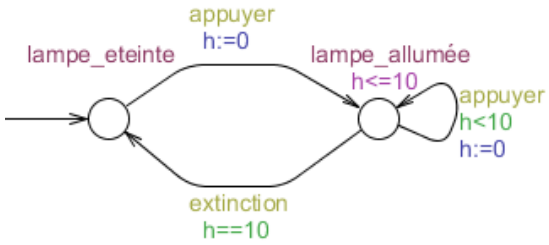


FIGURE 2.5 – Exemple d'automate temporisé

une condition portant sur des horloges. Les transitions ont également la possibilité de remettre à zéro une ou plusieurs horloges. Les différents types d'états pouvant être représentés sont l'état simple, l'état initial et l'état acceptant. Chaque état peut également posséder un invariant portant sur une ou plusieurs horloges.

Un exemple d'automate temporisé est disponible Figure 2.5. Il décrit le fonctionnement d'une lampe équipée d'une minuterie. Dans l'état `lampe_eteinte`, l'action `appuyer` permet de passer dans l'état `lampe_allumée` et de mettre l'horloge `h` à 0. Dans l'état `lampe_allumée`, l'invariant `h` $\leq$ `10` indique que le système ne peut pas rester plus de 10 unités de temps dans cet état. Il peut soit franchir une transition avant ces 10 unités de temps grâce à l'action `appuyer`, ce qui remet l'horloge `h` à 0 et fait revenir le système dans l'état `lampe_allumée` ou le système doit passer dans l'état `lampe_eteinte` au bout de 10 unités de temps exactement, avec l'action `extinction`.

Outre la représentation de systèmes temps-réel, ces automates permettent de vérifier des propriétés temporelles sur un système. L'outil UPPAAL [52] [14] permet la modélisation, la simulation et la vérification de systèmes temps-réel. La modélisation est basée sur une extension des automates temporisés et les propriétés à vérifier sont exprimées en logique temporelle.

Les Statecharts

Dans un autre cadre que les systèmes temps-réel, les statecharts [70] sont un formalisme permettant la représentation comportementale de systèmes réactifs [53]. Le formalisme qui est proposé repose sur une extension des concepts classiques d'une machine à états. Un statechart est composé d'un ensemble fini d'états et de transitions. Une transition est composée d'un événement défini comme *trigger* [54] et/ou d'une garde, qui est une condition booléenne de franchissement de la transition. Une transition peut également définir une action, exécutée instantanément au franchissement de la transition.

Concernant les états, les statecharts offrent des possibilités autres que la représentation d'un état simple et d'un état initial. Ils permettent notamment la représentation

52. `http://www.uppaal.org/`
53. Un système réactif est un système qui répond à des stimuli internes ou externes, en évoluant et/ou en produisant des actions.
54. En français : gâchette.

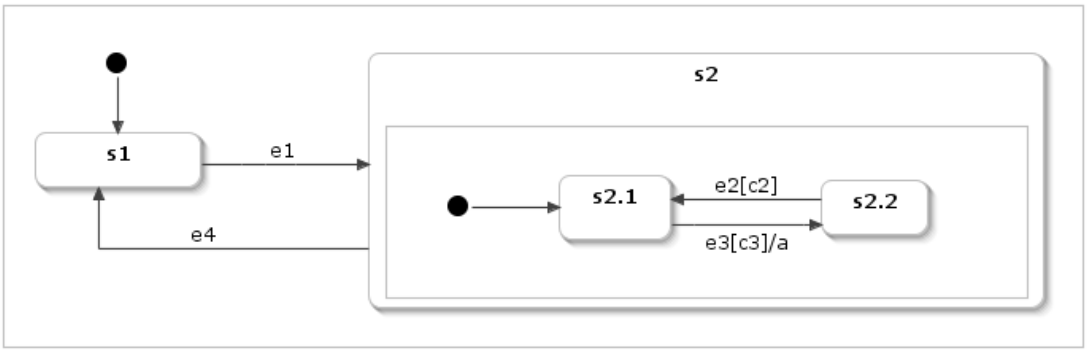


FIGURE 2.6 – Exemple de statechart

d'états hiérarchiques, dit "composés". Un état composé peut contenir un comportement décrit à l'aide d'états et de transitions. Il peut également être décomposé en plusieurs partitions, représentant chacune un comportement exécuté en parallèle. Cette décomposition modulaire permet de factoriser les comportements et ainsi améliorer la lecture et réduire la complexité des machines à états.

Un statechart fonctionne par une succession de pas. Chaque pas permet de décider quelles transitions peuvent être franchies, quelles sont les actions à exécuter et les nouveaux états courants du statechart. Un exemple de statecharts est donné Figure 2.6. Cet exemple est composé d'un état initial, d'un état simple (`s1`) et d'un état composé (`s2`) qui possède 2 sous-états (`s2.1` et `s2.2`). Le passage de `s1` à `s2` est conditionné par l'événement `e1`. A l'entrée dans `s2`, la machine à états entre dans `s2.1`. Le passage de `s2.1` à `s2.2` est conditionné par l'événement `e3` et la condition `c3` et entraine l'exécution de l'action `a`. Le passage de `s2.2` à `s2.1` est conditionné par l'événement `e2` et la condition `c2`. Il est possible de sortir de l'état `s2`, et donc de n'importe quel de ses sous-états, par la transition conditionnée par l'événement `e4`.

Pour un aperçu plus exhaustif, le lecteur pourra consulter l'ouvrage [72].

Les machines à états UML

Le standard de modélisation UML [98] définit un formalisme de machine à états pour la représentation comportementale de logiciels. Ces machines à états sont une variante du formalisme de statecharts précédemment décrit. Nous utiliserons les termes du standard pour décrire ces machines à états.

Comme toute machine à états, les machines à états UML sont constituées d'états et de transitions. Une transition entre deux états peut posséder trois attributs : un *Trigger* qui est associé à la détection d'un événement et qui permet de déclencher une transition ; une *Guard* qui est une expression booléenne qui permet le franchissement de la transition ; et un *Effect* qui permet de définir des actions sur la transition, exécutables à son franchissement.

Un état peut posséder des activités qui représentent les actions effectuées dans cet

état. Dans le standard UML, ces activités peuvent être spécifiées de trois manières différentes : en *Entry*, l'activité est alors exécutée à l'entrée de l'état jusqu'à sa terminaison ; en *Exit*, l'activité est exécutée à la sortie de l'état jusqu'à sa terminaison ; ou en *DoActivity*, l'activité est dans ce cas exécutée au cours de l'état, jusqu'à sa terminaison ou jusqu'à la sortie de l'état.

Le standard UML définit également des éléments appelés des *Pseudostate*[55]. Ces *Pseudostate* permettent de représenter des nœuds transitoires au sein de la machine à états, qui ne peuvent pas être considérés comme des états mais qui restent nécessaires à la modélisation. Parmi ces *Pseudostate*, nous pouvons citer à titre d'exemple l'*Initial pseudostate* qui représente l'état initial, ou encore le *Choice Pseudostate* qui représente un nœud de décision pour les transitions.

Le comportement des machines à états UML est géré par les événements. Chaque machine à états dispose d'un espace de stockage des événements appelé *pool*, la politique d'entrée/sortie de la *pool* étant définie par l'utilisateur. Les événements sont traités par la machine à états suivant la notion de *Run-to-Completion*. Quand un événement est pris dans la *pool*, si cet événement permet de franchir une transition, en accord avec sa condition de garde, alors il est consommé ; sinon, l'événement est simplement jeté. La notion *Run-to-Completion* précise qu'une occurrence d'événement ne peut être traitée que si le traitement de la précédente occurrence est terminé. Le traitement d'un événement est alors appelé *Run-to-completion step*. Ce traitement représente le passage entre deux configurations stables d'états de la machine à états. Une machine à états est dans une configuration stable d'états lorsqu'elle se trouve dans un état et qu'aucune activité n'est en cours (hormis les activités définies en *DoActivity*).

Parmi les événements traités par une machine à états UML, il existe un événement qui est défini implicitement dans le standard : le *completion event*. Le *completion event* est un événement généré automatiquement lorsque tous les comportements d'un état ont été traités et il est prioritaire sur tous les autres événements de la *pool*. Ainsi, si une transition sortante d'un état ne possède pas de *Trigger* spécifique, le standard considère que son *Trigger* est défini par défaut par l'événement *completion event* et le déclenchement de la transition est automatique.

Un exemple est donné Figure 2.7. Il représente le fonctionnement très simplifié d'un photocopieur. A son lancement, le photocopieur passe dans un état `Initialisation` dans lequel il exécute, à son entrée, l'initialisation de ces paramètres grâce à l'activité `initialiser_paramètres`. A la fin de cette activité, il passe automatiquement dans l'état `Repos`, grâce à la transition définie sans *Trigger* et sans *Guard*. Dans l'état `Repos`, si le photocopieur reçoit un événement `Appuyer` indiquant la demande d'une photocopie, il franchit la transition le menant à l'état `Photocopie` à condition qu'il y ait des feuilles disponibles dans le bac ; sinon, il reste dans l'état `Repos`. Dès l'entrée dans l'état `Photocopie`, le photocopieur exécute l'activité `Imprimer_copie` jusqu'à sa terminaison puis revient automatiquement dans l'état `Repos`.

Pour finir, UML définit également des états hiérarchiques pour la factorisation des comportements, similairement aux statecharts. Dans ce cadre, le standard distingue deux

55. En français : pseudo-état.

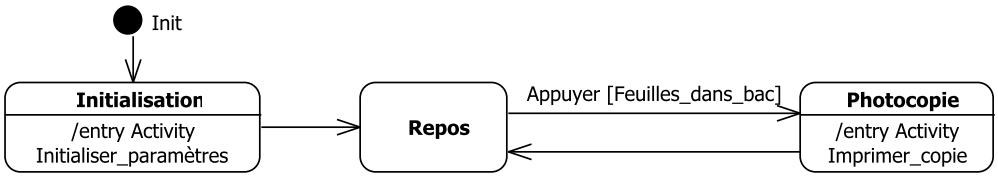


FIGURE 2.7 – Exemple de machine à états UML

types d'états en addition aux états simples : le *CompositeState* et le *SubmachineState*. Le *CompositeState* est un état englobant un comportement décrit à l'aide d'états, de transitions et de *Region*. Sommairement, une *Region* représente un conteneur d'états et de transitions. Un *CompositeState* peut contenir plusieurs *Region* dites orthogonales : les comportements décrits dans ces différentes *Region* sont alors exécutés en parallèle dès l'entrée dans le *CompositeState*. Notons qu'une machine à états possède implicitement une *Region* contenant tous ses états et transitions et qu'il existe un état particulier nommé *FinalState* en UML qui permet de représenter l'achèvement du comportement décrit dans une *Region*. En comparaison, le *SubmachineState* permet de faire référence à une machine à états externe. La communication entre le *SubmachineState* et la machine à états référencée peut se faire via deux types de *Pseudostate* : un *EntryPoint* qui représente un état d'entrée dans la machine à états référencée et un *ExitPoint* qui correspond à un état de sortie. Ces *Pseudostate* sont ensuite associés à des *ConnectionPointReference* sur le *SubmachineState*.

Les concepts qui viennent d'être présentés représentent les concepts principaux des machines à états UML. Nous avons volontairement passé sous silence une partie des concepts de la norme pour rendre la lecture plus didactique. Pour une description exhaustive, le lecteur pourra consulter le standard [98].

Autres automates

Il existe d'autres formalismes d'automates pour la représentation de systèmes, tels que les systèmes d'états-transitions [7], les machines à états Rhapsody UML [71] ou encore les réseaux de Petri [104]. Cependant, nous ne décrirons pas davantage de formalismes, l'aperçu qui a été donné étant suffisant pour poursuivre la thèse.

2.3 La vérification de programmes

Nous avons abordé l'IDM et la modélisation comportementale via le concept de machine à états. Il nous reste à introduire dans cette thèse la thématique de la vérification de programmes.

Il existe plusieurs approches pour vérifier un programme. La plus classique consiste

à soumettre le programme à des tests, la confiance dans ces tests étant assurée par leur taux de couverture des scénarios d'exécution possibles du logiciel. Une autre approche est d'employer les méthodes formelles afin de vérifier formellement le programme. Le standard de certification avionique DO-178C donne la définition suivante des méthodes formelles : *"Descriptive notations and analytical methods used to construct, develop, and reason about mathematical models of system behavior. A formal method is a formal analysis carried out on a formal model."*[56]. L'emploi des méthodes formelles permet d'obtenir une plus grande confiance dans la vérification par rapport aux tests, mais elles requièrent en contre-partie un plus grand investissement en terme de temps et de connaissances techniques de la part de l'utilisateur.

2.3.1 La vérification par tests

Les méthodes de tests

Il existe deux grands types de stratégies pour conduire des tests [93] : le test dit en boîte noire et le test dit en boîte blanche. Le test boîte noire considère le programme en fonction de ses entrées et de ses sorties. Il ne s'intéresse pas à la structure ou aux traitements internes du programme. Dans ce cadre, il permet de vérifier qu'un programme se comporte comme décrit dans sa spécification suivant des valeurs d'entrée. Cependant, il est impossible de tester exhaustivement un programme en fonction de ses valeurs possibles d'entrée. A titre d'exemple, prenons le cas d'un simple programme prenant en entrée 2 entiers. Pour un test exhaustif, il faudrait écrire et tester l'ensemble des combinaisons possibles de ces entrées sur l'ensemble des entiers. Si nous prenons une version de ce programme écrite dans le langage Java, les entiers allant de $2^{31} - 1$ à -2^{31}, il existe plus de 4000000000^2 de combinaisons possibles à tester. En considérant qu'il faut 1 milli-seconde pour écrire le test d'une combinaison, l'exécuter et vérifier le résultat, il faudrait approximativement 10 milliards d'années pour tester exhaustivement ce simple programme pour toutes ses valeurs d'entrée.

Le test boîte blanche considère la structure interne du programme. A la différence du test boîte noire, il permet de tester un programme par rapport à sa logique interne afin de vérifier qu'il ne comporte pas d'erreurs. Les tests vont alors porter sur les chemins de flots de contrôle possibles pouvant être empruntés dans le programme. Cependant, cette méthode de tests ne permet pas pour autant de tester exhaustivement le programme. De la même manière qu'il faudrait tester l'ensemble des valeurs d'entrée pour un test boîte noire, il faudrait tester l'ensemble des chemins de flots de contrôle possible pour pouvoir être exhaustif dans le cadre d'un test boîte blanche. Ce nombre de chemins peut être déraisonnablement grand, notamment pour des programmes comprenant des boucles itératives.

[93] rappelle que, même s'il était possible d'être exhaustif à partir des tests boîte blanche, il n'y aurait pas de garantie que le programme soit correct. En effet, le test

56. En français : Notations descriptives et méthodes analytiques permettant de construire, développer et raisonner sur des modèles mathématiques du comportement d'un système. Une méthode formelle est une analyse formelle effectuée sur un modèle formel.

exhaustif des chemins d'un programme ne permet pas de vérifier s'il manque des chemins nécessaires à ce programme, ce qui compromet la vérification du programme par rapport à une spécification. Dans ce contexte, une méthode de test efficace peut alors se baser sur une combinaison de tests boîte noire et de tests boîte blanche.

Les stratégies de définition de tests

La définition des tests ne pouvant être exhaustive quelle que soit la méthode de tests préconisée, la fiabilité des tests dépend nécessairement des cas de tests qui vont être définis et de leur taux de couverture des erreurs. Dans ce contexte, le but n'est pas de définir des cas de tests qui vont confirmer que le programme fonctionne correctement, mais de définir des cas de tests qui vont permettre de détecter les défaillances d'un programme. Il existe différentes stratégies pour définir des cas de tests. [93] recense la plupart d'entre elles. A titre d'exemple, nous en énumérons quelques-unes pour chaque méthode de test.

Concernant la définition de tests boîte noire, la stratégie dite d'*"error guessing"*[57] est l'une des plus intuitives. Elle se base principalement sur l'expérience des personnes qui définissent les tests. Le principe est d'écrire des tests afin de détecter des erreurs qui peuvent intuitivement se produire ou connues pour se produire dans le type de programme testé. Un autre exemple est la stratégie de définition dite d'*"equivalence partitioning"*[58]. Elle permet de définir des tests en fonction de classes d'équivalence. Une classe d'équivalence est une partition des valeurs d'entrée possibles du programme. Une classe regroupe toutes les valeurs susceptibles d'engendrer les mêmes erreurs. Dans ce sens, si un cas de test défini à partir de valeurs d'une classe d'équivalence détecte une erreur, n'importe quel autre cas de test défini à partir d'autres valeurs de la même classe doit détecter cette même erreur. Ces classes d'équivalence sont principalement déduites à partir d'informations fournies dans la spécification.

Concernant la définition de tests boîte blanche, la stratégie dite de *"decision coverage"*[59] permet de définir des cas de tests en fonction des décisions présentes dans le programme. Une décision est une expression booléenne formée de conditions booléennes liées entre elles par des opérateurs booléens (tels que la conjonction ou la disjonction). Un cas de test est défini pour chaque branche de la décision : un pour couvrir le cas où elle renvoie vrai, un autre pour couvrir le cas où elle renvoie faux. Une autre stratégie dite de *"condition coverage"*[60] permet elle de définir un cas de test pour couvrir chaque valeur possible pour chaque condition booléenne du programme.

De nombreux autres travaux ont porté sur la définition de tests. Par exemple, [84] aborde la spécification de cas de test par définition d'objectifs de test. Un objectif de test représente une abstraction de cas de test et il permet d'obtenir, par raffinement, des cas de test pour répondre à une exigence ou à un besoin précis.

57. En français : prévision des erreurs.
58. En français : partitionnement par équivalence.
59. En français : couverture par les décisions.
60. En français : couverture par les conditions.

Les étapes de tests dans le processus de développement

Dans un développement logiciel classique, la phase de test se découpe en plusieurs étapes. Cette phase passe tout d'abord par l'étape de test la plus connue, l'étape des tests unitaires. Les tests unitaires permettent de vérifier unitairement chaque module d'un logiciel, indépendamment des autres. La définition d'un test unitaire peut cependant varier selon les sources. Selon le glossaire standard des termes liés à l'ingénierie du logiciel défini par IEEE[61] [1], un test unitaire est défini comme : *"testing of individual units or groups related units"*[62]. En comparaison, [93] définit le test unitaire comme *"a process of testing the individual subprograms, subroutines, or procedures in a program. That is, rather than initially testing the program as a whole, testing is first focused on the smaller building blocks of the program."*[63]. D'une manière plus empirique, [115] présente une étude sur la pratique des tests unitaires réalisée auprès de plusieurs entreprises.

Les retours de son étude montrent qu'un test unitaire est considéré comme *"testing the smallest separate module in the system"*[64]. La plupart de ces définitions expriment la même idée. Cependant comme [115] le fait remarquer, les opinions peuvent diverger sur le fait que les unités soient issues des spécifications ou définies par les développeurs. La réalisation de tests unitaires présente plusieurs avantages dans un processus de développement logiciel. Dans son étude, [115] rapporte que les industriels interrogés y voient un moyen de tester la sûreté de fonctionnement d'un logiciel. [93] explique que le test unitaire permet notamment de tester simultanément plusieurs modules en parallèle. Il permet également de faciliter la tâche de débuggage puisque lorsqu'une erreur est trouvée, elle est située dans une unité connue du programme.

Suite à l'étape de tests unitaires, la phase de test passe par une étape de tests d'intégration afin de vérifier que les différentes unités, ou modules, du programme fonctionnent correctement entre eux. Selon le glossaire standard des termes liés à l'ingénierie du logiciel défini par IEEE [1], le test d'intégration est défini comme : *"Testing in which software components, hardware components, or both are combined and tested to evaluate the interaction between them."*[65].

Les tests unitaires et les tests d'intégration sont les deux principaux tests permettant de détecter des erreurs de programmation. Suite à ces deux étapes, d'autres types de tests sont réalisables, tels que les tests de non-régression, d'installation, de performance, etc, mais nous considérons que ces types de tests sortent du cadre de la détection d'erreurs de programmation et ne sont donc pas décrits ici.

61. Institute of Electrical and Electronics Engineers.
62. En français : le test d'unités individuelles ou de groupes d'unités liées.
63. En français : un processus pour tester les sous-programmes ou procédures individuelles d'un programme. Il s'agit en fait, plutôt que de commençer par tester le programme dans sa globalité, de tester tout d'abord les plus petits composants d'un programme.
64. En français : tester le plus petit module unitaire d'un système.
65. En français : test dans lequel des composants logiciels ou physiques, ou les deux, sont combinés et testés afin d'évaluer l'interaction entre eux.

2.3.2 Les vérifications formelles statiques

Parmi les méthodes formelles dédiées à la vérification de programmes, il émerge deux grandes catégories de méthodes : les méthodes de vérification statique et les méthodes de vérification dynamique.

Les méthodes de vérification statique sont des méthodes permettant de vérifier un programme sans l'exécuter. Par définition, la vérification de la correction d'un programme est indécidable. Il n'existe aucune machine capable de statuer systématiquement dans un temps fini si un programme contient des erreurs ou non. Cependant, les méthodes de vérification statique proposent des solutions pour contourner ce problème d'indécidabilité et réaliser des preuves sur une abstraction du programme. A la différence des méthodes de vérification par test, les méthodes de preuve vont permettre de vérifier exhaustivement un programme par rapport à une spécification.

Plusieurs techniques existents. Nous allons décrire deux d'entre elles : l'interprétation abstraite et les méthodes déductives. Notons qu'il existe également des techniques de vérification de programmes par model-checking ([74] par exemple), mais nous n'aborderons pas ces travaux.

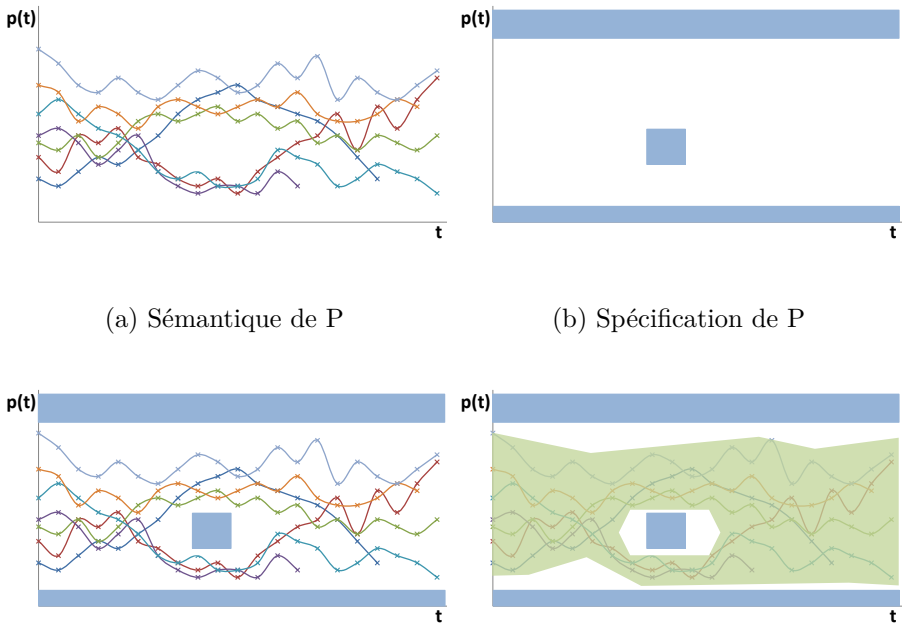
L'interprétation abstraite

Afin de pallier le problème d'indécidabilité précédemment évoqué, ou tout simplement afin de s'abstraire de la complexité d'un programme, une méthode envisageable est d'approximer la sémantique d'un programme. Cette approximation se présente sous la forme d'une perte d'informations, et donc de précision, qui ne sont pas nécessaires au but poursuivi. Par exemple dans le cadre d'une multiplication entre deux entiers, si le but est seulement de connaître le signe du résultat de cette multiplication, nous pouvons nous abstraire des valeurs des entiers pour se concentrer uniquement sur leurs signes.

L'interprétation abstraite est une méthode, mise au point par Patrick et Radhia Cousot durant les années 70 [29, 27], qui a pour but de donner des bases formelles à l'approximation de la sémantique des langages. Dans [30], ils définissent l'interprétation abstraite comme : *"a theory of approximation of mathematical structures, in particular those involved in the semantic models of computer systems. Abstract interpretation can be applied to the systematic construction of methods and effective algorithms to approximate undecidable or very complex problems in computer science such that the semantics, the proof, the static analysis, the verification, the safety and the security of software or hardware computer systems."* [66].

Appliquée à la vérification de programmes, cette méthode va permettre d'approximer la sémantique du programme pour réaliser la preuve de sa correction par rapport à une spécification. Pour illustrer cette méthode, prenons un exemple simple inspiré de [28].

66. En français : une théorie de l'approximation des structures mathématiques et plus particulièrement de celles impliquées dans les modèles sémantiques des systèmes informatiques. L'interprétation abstraite peut être appliquée à la définition de méthodes ou d'algorithmes optimaux pour l'approximation de problèmes indécidables ou très complexes en informatique tels que la sémantique, la preuve, l'analyse statique, la vérification, la sûreté de fonctionnement et la sécurité de logiciels ou de matériels informatiques.



(a) Sémantique de P

(b) Spécification de P

(c) Objectif de la preuve de P

(d) Approximation de P

FIGURE 2.8 – Illustration de l'interprétation abstraite

Un programme est composé de pas de calculs, chacun correspondant à des états atteignables du programme. Les différents comportements d'un programme peuvent alors être représentés comme un chemin à travers ces états au cours du temps. Considérons la sémantique d'un programme P comme l'ensemble de ces chemins possibles comme décrit Figure 2.8a. La spécification du programme définit les chemins autorisés du programme ainsi que ses chemins interdits (par exemple, une division par zéro). La spécification de notre programme P est représentée Figure 2.8b. La vérification du programme consiste à vérifier que l'ensemble des chemins possibles du programme respectent la spécification, c'est à dire que la sémantique du programme est incluse dans la spécification comme décrit Figure 2.8c. L'interprétation abstraite se base sur le calcul d'une sur-approximation de la sémantique de P comme décrit Figure 2.8d, qui va simplifier sa vérification par rapport à la spécification. Le calcul de cette sur-approximation peut notamment se faire par le calcul d'invariants sur le programme.

L'un des principes de l'interprétation abstraite est d'être correcte : la méthode ne reste pas muette face à des possibles erreurs mais elle peut retourner des fausses alarmes, c'est à dire détecter des erreurs qui n'en sont pas. Ces fausses alarmes sont alors dues à une sur-approximation trop grande de la sémantique du programme.

Les méthodes de preuve par interprétation abstraite sont aujourd'hui implémentées

dans des outils et utilisées industriellement. Par exemple, l'outil Astrée [67] a notamment été utilisé par Airbus pour la preuve d'absence d'erreurs à l'exécution pour des programmes de commandes de vol électriques sur l'A380. Un autre exemple est l'outil CGS(C global Surveyor) [68] développé et utilisé par la NASA. Il a notamment été utilisé pour la vérification de logiciels de vols pour les engins spatiaux Mars Path-Finder, Deep Space One et Mars Exploration Rover. Il existe également l'outil Polyspace [69] mis au point par Mathworks qui permet la détection automatique d'erreurs et qui est utilisé dans les domaines automobile, ferroviaire et aéronautique. Enfin, citons le greffon Value Analysis [70] de l'outil open-source Frama-C [71] pour la vérification formelle de code C ainsi que les analyseurs statiques proposés par la société AbsInt [72].

Les méthodes déductives et la logique de Hoare

Les méthodes déductives sont un autre type de méthodes envisageables pour conduire une preuve de programme. Elles consistent à prouver par déduction des assertions sur un programme en se basant sur un ensemble d'axiomes et de règles d'inférences préalablement définies. Dans [76], Hoare pensait que les propriétés d'un programme, ainsi que les conséquences de son exécution, pouvaient être directement déduites à partir du "texte" de ce programme. Il pose alors les bases logiques pour formaliser ce type de raisonnement. Ces travaux s'intéressent principalement à vérifier le comportement attendu d'un programme, en s'appuyant à la fois sur les valeurs des variables du système à l'entrée du programme, sur les valeurs de ces variables à la sortie du programme et sur le corps du programme lui-même. Ses travaux s'inspirent de ceux de Floyd [59] qui proposait également de raisonner sur la vérification d'un programme en considérant des propriétés sur ses variables à l'état initial et à l'état final.

La logique de Hoare, ou logique de Floyd-Hoare, se base sur une notation très simple pour représenter un programme et les propriétés à vérifier sur ce programme sous forme d'assertions. Cette notation se nomme le triplet de Hoare. Ce triplet est défini dans la Définition 1.

Définition 1 (Triplet de Hoare)**.** *Soit P et R des assertions et $Prog$ un programme. Un triplet de Hoare est noté :*

$$P\{Prog\}R$$

Son interprétation est la suivante : "Si l'assertion P est vraie avant le début de l'exécution du programme $Prog$, alors l'assertion R sera vraie si $Prog$ se termine". P est appelée une précondition et R est appelée une postcondition.

67. www.astree.ens.fr
68. http://ti.arc.nasa.gov/tech/rse/vandv/cgs/
69. www.mathworks.fr/products/polyspace
70. frama-c.com/value.html
71. frama-c.com
72. www.absint.com

Prenons un exemple simple : considérons `x` une variable entière définie dans un langage de programmation. Le triplet $(x > 0)$`{x = x × (-1);}`$(x < 0)$ est valide selon la logique de Hoare.

En s'appuyant sur cette notation sous forme de triplet, la logique de Hoare définit un ensemble d'axiome et de règles d'inférence afin de pouvoir raisonner par déduction. L'axiome principal de cette logique porte sur l'instruction la plus basique qu'il est possible de trouver dans un programme : l'affectation. Sa définition est donnée dans la Définition 2.

Définition 2 (Axiome d'affectation)**.** *Soit x une variable, f une expression sans effet de bords du programme mais qui peut contenir x, P et P_0 des assertions telles que P_0 est obtenue de P en substituant f à toutes les occurrences de x dans P. L'axiome d'affectation peut alors être exprimé comme :*

$$\vdash P_0\{x := f\}P$$

En d'autres termes, si la propriété P est vraie après l'affection de x, c'est à dire que $P(x)$ est vraie, alors $P_0 = P(f)$ est vraie avant l'affectation.

La logique de Hoare définit ensuite trois règles d'inférence pour pouvoir raisonner par déduction. La première est la règle de conséquence définie dans la Définition 3. La deuxième est la règle de composition énoncée dans la Définition 4. La dernière est la règle d'itération qui permet de raisonner sur les boucles d'un programme. Sa définition est donnée dans la Définition 5.

Définition 3 (Règle de conséquence)**.** *Soit P , R et S des assertions et $Prog$ un programme. La règle de conséquence définit :*

$$\textbf{\textit{si}} \vdash P\{Prog\}R \textbf{\textit{ et}} \vdash R \supset S \textbf{\textit{ alors}} \vdash P\{Prog\}S$$
$$\textit{et}$$
$$\textbf{\textit{si}} \vdash P\{Prog\}R \textbf{\textit{ et}} \vdash S \supset P \textbf{\textit{ alors}} \vdash S\{Prog\}R$$

En d'autres termes, si l'exécution d'un programme $Prog$ garantit la postcondition R à sa terminaison à condition que la précondition P soit vraie alors toutes les assertions impliquées par R sont également garanties. Similairement, si la précondition P permet de garantir la postcondition R après l'exécution de $Prog$, alors il en est de même pour toutes les assertions qui impliquent P.

Définition 4 (Règle de composition)**.** *Soit $Prog_1$ et $Prog_2$ deux portions d'un même programme et P, R_1, R des assertions. La règle de composition définit :*

$$\textbf{\textit{si}} \vdash P\{Prog_1\}R_1 \textbf{\textit{ et}} \vdash R_1\{Prog_2\}R \textbf{\textit{ alors}} \vdash P\{Prog_1; Prog_2\}R$$

En d'autres termes, si la précondition P garantit la postcondition R_1 de la première partie $Prog_1$ d'un programme et que R_1 garantit une postcondition R pour l'exécution de $Prog_2$, alors la précondition P garantit la postcondition R après l'exécution successive de $Prog_1$ et $Prog_2$.

Définition 5 (Règle d'itération)**.** *Soit P une assertion, B une condition booléenne et Subprog une portion d'un programme. La règle d'itération définit :*

$$\textbf{\textit{si}} \vdash P \wedge B\{Subprog\}P \textbf{\textit{ alors}} \vdash P\{\textbf{\textit{tant que}}\ B\ \textbf{\textit{faire}}\ Subprog\}\neg B \wedge P$$

En d'autres termes, si P est vraie avant l'exécution de Subprog et est toujours vraie après son exécution, alors P sera toujours vraie après n'importe quel nombre d'itérations de Subprog. La condition booléenne de contrôle B est supposée vraie au lancement de l'itération.

Hoare a établi d'autres règles suite à [76] mais nous ne les décrirons pas dans cette thèse, les règles de base étant suffisantes pour éclairer le lecteur sur l'approche. La logique de Hoare se veut correcte et complète à condition que la logique sous-jacente du langage d'assertions le soit également, ce qui est le cas pour la logique classique du premier ordre. Cependant la logique de Hoare ne permet pas de prouver la terminaison du programme. Les règles précédemment décrites sont donc applicables "à condition que le programme vérifié se termine".

Cette logique a permis de définir une base formelle pour la preuve par déduction d'un programme. Dans [39], Dijkstra se base sur cette logique pour établir une approche plus automatisable. Il propose de calculer la précondition la plus faible garantissant la postcondition portant sur un programme. Cette précondition la plus faible est définie dans la Définition 6.

Définition 6 (Précondition la plus faible)**.** *Soit Prog un programme et R une assertion représentant une postcondition sur ce programme. La précondition la plus faible garantissant la postcondition R est notée :*

$$wp(Prog, R)\ \textbf{\textit{tel que}} \vdash wp(Prog, R)\{Prog\}R$$

Le calcul de cette précondition la plus faible se nomme communément "*weakest precondition calculus*" [73]. Pour un triplet de Hoare $P\{Prog\}R$, ce calcul est réalisé en partant de la postcondition et en remontant par chacune des instructions du programme jusqu'à son commencement. Une fois ce calcul effectué, il suffit alors de prouver que $P \implies wp(Prog, R)$ pour prouver $P\{Prog\}R$ par application de la règle de conséquence. Cette implication se limitant à une formule logique et étant complètement indépendante du corps du programme, elle peut être automatiquement prouvée par des solveurs. Ce calcul de la plus faible précondition fonctionne très bien sur des programmes simples mais devient non automatique pour des programmes comprenant des boucles.

Ce problème de boucle peut alors être contourné en ajoutant manuellement des invariants de boucles qui vont faciliter le calcul. A l'aide de ces invariants, il suffit alors de calculer des conditions de vérification (VC [74]). La définition d'une condition de vérification est donnée dans la Définition 7.

73. En français : calcul de la plus faible précondition.
74. En anglais : verification condition.

Définition 7 (Condition de vérification)**.** *Soit Prog un programme et R une assertion représentant une postcondition sur ce programme et wp(Prog, R) la plus faible précondition sur Prog garantissant R. Une condition de vérification est notée :*

$$VC(Prog, R) \text{ tel que} \vdash VC(Prog, R) \implies wp(Prog, R)$$

Pour un triplet de Hoare $P\{Prog\}R$, il suffit alors de prouver $P \implies VC(Prog, R)$ pour prouver $P\{Prog\}R$. Il est également possible de rajouter des variants de boucles qui sont des propriétés portant sur l'évolution d'une ou plusieurs variables au sein de la boucle et qui vont servir à statuer sur sa terminaison, et donc sur la terminaison du programme.

Les outils actuels de preuve statique par déduction s'appuient sur ce genre de techniques. A partir de préconditions et de postconditions définies par l'utilisateur sur le programme, ainsi que d'invariants et de variants de boucles définis si nécessaire, ces outils calculent automatiquement un ensemble de conditions de vérification et délèguent ensuite la preuve à des outils de preuve (ou démonstrateurs automatiques). Un certain nombre de ces outils existent aujourd'hui et certains sont utilisés industriellement. A titre d'exemple, nous pouvons citer CAVEAT [13], un outil développé par le CEA qui permet de réaliser de l'analyse statique sur un programme C à partir des méthodes déductives. Il a notamment été utilisé par Airbus et EDF. Citons également l'outil Why3 [75] [57], développé par l'INRIA Saclay-Île-de-France, LRI Univ Paris-Sud 11 et le CNRS. Il s'agit d'une plate-forme pour la preuve de programme par déduction. Enfin, la plate-forme Frama-C [76] est une plate-forme open-source pour l'analyse statique de code C. Son greffon WP [77] se base sur les méthodes déductives pour la preuve de programme et peut notamment être associé à l'outil Why3 pour faire le lien avec des solveurs externes.

2.3.3 Les vérifications formelles dynamiques

Outre l'analyse statique, la vérification formelle d'un programme peut également s'appuyer sur une analyse dynamique. Les méthodes dites "dynamiques" sont des techniques permettant de réaliser la vérification formelle de programme à l'aide de son exécution. Cette vérification peut être menée durant l'exécution, ou bien après l'exécution.

Vérifier formellement un programme pendant son exécution consiste à insérer dans le code des proporiétés sous forme d'assertions à vérifier à l'exécution. Si l'une de ces assertions n'est pas vérifiée, le programme s'arrêtera et retournera une erreur. Cette technique de vérification se nomme communément Runtime Assertion Checking [78]. Par exemple, les travaux de [83] présentent le langage d'assertion E-ACSL [79] et le greffon E-ACSL [80] de l'environnement Frama-C pour réaliser ce type de vérification sur des programmes C.

75. `http://why3.lri.fr/`
76. `http://frama-c.com/`
77. `http://frama-c.com/wp.html`
78. En français : vérification d'assertion à l'exécution.
79. Executable ANSI/ISO C Specification Language.
80. `frama-c.com/eacsl.html`

Le langage E-ACSL permet d'annoter le code C avec des assertions vérifiables à l'exécution. Le greffon E-ACSL associé peut ensuite transformer le programme annoté en un programme exécutable intégrant totalement ces assertions dans le code. L'exécution du programme créé permet alors de déterminer si les assertions sont vérifiées ou non. Ces assertions sont sans effet de bord sur le comportement original du programme.

Vérifier formellement un programme après son exécution consiste à analyser les traces d'exécution de ce programme. Ces traces vont permettre de vérifier le comportement du programme par rapport à des propriétés. Par exemple, les travaux de [48] portent sur l'étude de traces finies d'un programme afin de vérifier formellement des propriétés temporelles exprimées en Logique Temporelle Linéaire [81] (LTL). L'exécution du programme est réalisée sur une machine virtuelle et les traces sont obtenues à partir de points d'observation définis sur le programme à l'aide d'un débuggeur. La définition de ces points d'observation permet de sur-approximer les pas de calcul du programme et ainsi rendre l'analyse de la trace plus efficace. La trace obtenue est donc partielle mais [48] démontre que chaque propriété vérifiée sur la trace partielle d'exécution est également vérifiée sur la trace complète. La vérification est conduite en transformant la propriété LTL en un automate de Büchi puis en exécutant la trace sur cet automate pour vérifier la propriété.

2.3.4 Conclusion

Les méthodes de vérification dynamique, de la même manière que les méthodes de tests, s'appuient sur des exécutions du programme et ne permettent donc pas d'être aussi exhaustives que les méthodes de preuve statique qui tendent à vérifier le programme pour toute exécution possible. Cependant, les méthodes de preuve statique nécessitent une spécification formelle complexe et il n'est pas toujours évident pour un utilisateur d'arriver à une preuve exhaustive d'un programme. Les différents types de méthode peuvent alors être combinés pour combler leurs défauts respectifs et ainsi améliorer les processus de vérification actuels et la confiance portée aux logiciels produits.

81. En anglais : Linear Temporal Logic.

Chapitre 3

Contexte industriel et axe de recherche

Outre les thématiques abordées, nos travaux s'inscrivent dans un contexte industriel induit par le dispositif CIFRE de cette thèse. Il faut par conséquent tenir compte des facteurs propres à l'environnement entourant les activités de l'industriel Atos. Nous distinguons trois facteurs qui ont influencé notre axe de recherche et qui nous ont permis de définir un objectif précis à atteindre afin de repondre à notre problématique.

Le premier concerne les normes de certification. Dans le cadre des logiciels avioniques, ces normes définissent les contraintes de développement à suivre pour l'obtention de la certification du logiciel embarqué dans l'aéronef. La norme DO-178, établie par EUROCAE (European Organisation for Civil Aviation Equipment) et RTCA inc. est le standard de certification dédié à l'avionique. Sa version la plus récente, la DO-178C [112], a été publiée en 2011. Des extensions y ont été ajoutées pour tenir compte des progrès actuels dans les méthodes employées dans le développement logiciel. Ce standard est présenté Section 3.1.

Le second facteur concerne l'utilisation industrielle des méthodes formelles. Cette utilisation est principalement motivée par des attentes industrielles telles que la réduction des coûts et l'amélioration de la qualité. Les méthodes formelles sont souvent un moyen d'obtenir un haut niveau de qualité très tôt dans le cycle de développement et, par conséquent, un moyen de réduire les efforts de vérification dans les phases aval tels que les tests ou la maintenance. Des exemples d'utilisation et des observations réalisées au sein d'Atos sont présentées Section 3.2.

Le dernier facteur porte sur les technologies actuellement utilisées par les équipes d'Atos et une partie de leurs clients dans le cadre des activités d'IDM et de vérification formelle de code. Nous nous intéressons plus particulièrement au langage de modélisation UML et à l'outil de vérification formelle Frama-C. Des détails sur ces technologies sont donnés dans la Section 3.3.

En tenant compte de ces facteurs, nous définissons le problème ciblé par nos travaux et l'axe de recherche à suivre pour y répondre dans la Section 3.4.

3.1 La norme de certification pour les logiciels avioniques

3.1.1 DO-178C

Le but de la norme DO-178 n'est pas de définir ou d'imposer un cycle de développement particulier pour un logiciel avionique, mais d'identifier les objectifs à atteindre dans un cycle de développement afin d'obtenir la certification du logiciel.

Quatre différentes phases de développement sont identifiées dans le cadre des cycles de développement logiciel couverts par la norme DO-178C :

1. la phase de spécification [1] qui permet d'obtenir les exigences de haut-niveau [2] à partir des exigences système. Nous appellerons *spécification* ces exigences de haut-niveau dans nos travaux ;
2. la phase de conception [3] qui permet d'obtenir les exigences de bas-niveau [4] et l'architecture logicielle [5] à partir des exigences de haut-niveau. Nous appellerons *conception* ces exigences de bas-niveau dans nos travaux ;
3. la phase d'implémentation [6] qui permet de créer le code source [7] à partir des exigences de bas-niveau et de l'architecture logicielle ;
4. la phase d'intégration [8] qui permet d'embarquer le code exécutable [9] sur le matériel ciblé.

Les produits issus de ces quatre phases de développement doivent être vérifiés. Des objectifs de vérification sont donc définis pour chaque phase. La Figure 3.1 présente ces objectifs et les différents types d'activité réalisée pour les atteindre. La DO-178C distingue trois types d'activité pour la vérification : les revues, les analyses et les tests.

Les revues fournissent une évaluation qualitative de la correction d'un produit. Les analyses offrent des évaluations reproductibles de la correction d'un produit. Les revues et les analyses sont utilisées pour tous les objectifs de vérification concernant les exigences de haut-niveau, les exigences de bas-niveau, l'architecture logicielle et le code source. Par exemple sur la Figure 3.1, les vérifications de la conformité et de la traçabilité du code source par rapport aux exigences de bas-niveau sont réalisées à partir de revues et d'analyses. Les tests sont utilisés pour vérifier que le code exécutable est conforme aux exigences de bas-niveau et de haut-niveau, comme nous pouvons le voir sur la Figure 3.1.

3.1.2 DO-333

L'extension DO-333 [113] de la norme DO-178C détaille l'utilisation possible des méthodes formelles à chaque phase du développement et les chemins de vérification

1. Désignée par *"software requirements process"* dans la DO-178C.
2. En anglais : High Level Requirements (HLR).
3. Désignée par *"software design process"* dans la DO-178C.
4. En anglais : Low Level Requirements (LLR).
5. En anglais : software architecture.
6. Désignée par *"software coding process"* dans la DO-178C.
7. En anglais : source code.
8. Désignée par *"software integration process"* dans la DO-178C.
9. En anglais : executable object code.

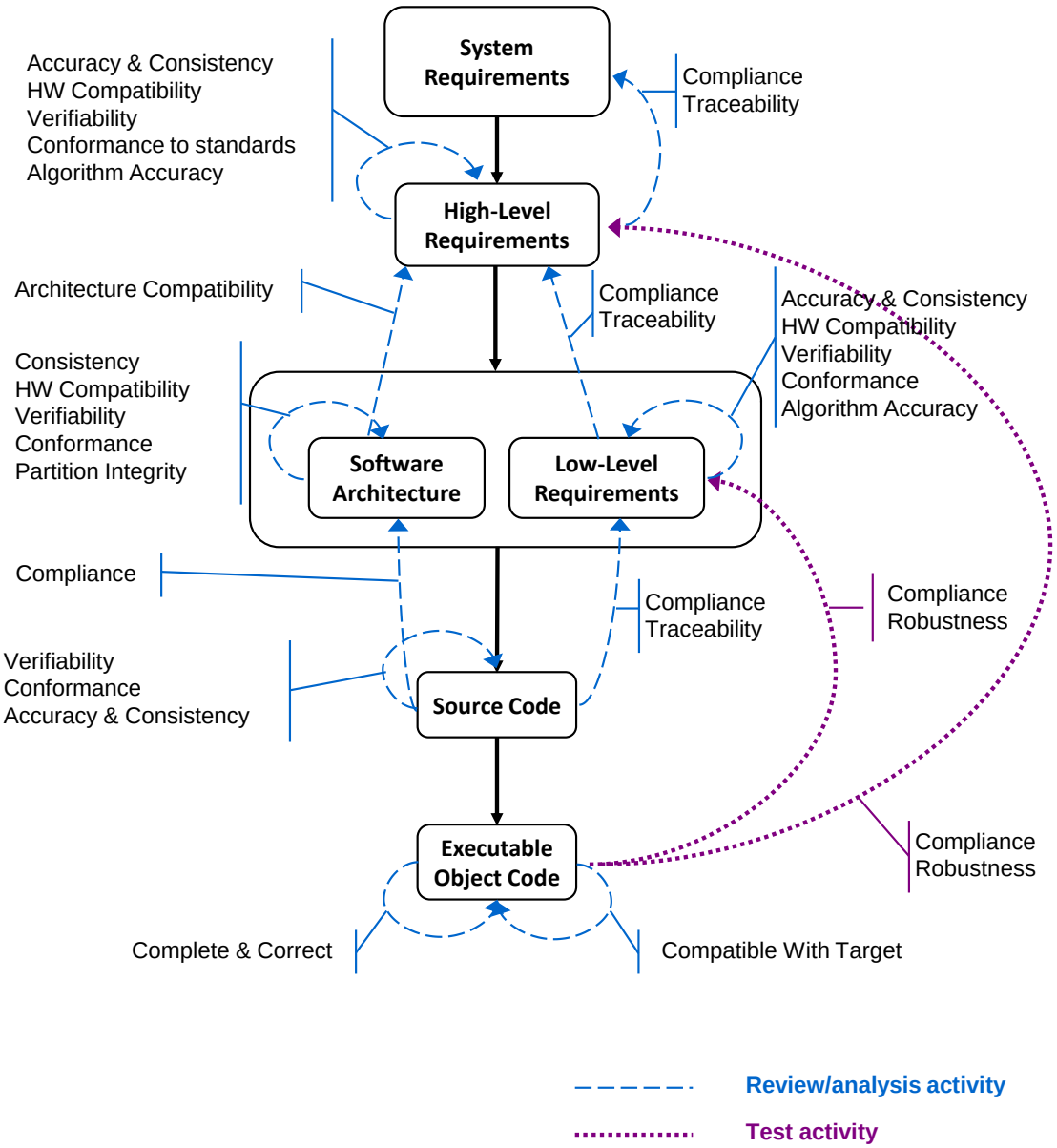


FIGURE 3.1 – Activités et objectifs de vérification dans le cadre de la DO-178C

alternatifs que cette utilisation implique par rapport au test. Une méthode formelle est une analyse formelle menée sur un modèle formel. Cette définition permet de discuter

des méthodes formelles selon les différentes phases des cycles de développement couverts par la DO-178, et particulièrement pour les phases de développement et de vérification. Les phases de développement peuvent alors s'appuyer sur des modèles formels et les phases de vérification peuvent s'appuyer sur des analyses formelles.

Avec une analyse formelle, la correction d'une donnée utilisée dans un cycle de développement par rapport à un modèle formel ou à une propriété peut généralement être prouvée ou invalidée. Par conséquent, une analyse formelle est apte à remplacer les activités de revues, d'analyses et de tests spécifiés dans la DO-178C pour la vérification de certains des objectifs définis.

3.2 Utilisation industrielle des méthodes formelles et observations à Atos

Exemples d'utilisation

Les méthodes formelles sont déjà présentes dans l'industrie. Par exemple, [121] présente une application industrielle de techniques de vérification formelle pour des logiciels avioniques à Airbus. Plus précisément, les auteurs les utilisent pour prouver des propriétés sur le code, pour analyser les pires cas de temps d'exécution [10] et pour le calcul de la consommation mémoire. Les auteurs présentent également les techniques formelles qu'ils prévoyaient d'utiliser dans des projets futurs telles que la preuve d'absence d'erreurs à l'exécution et qui font aujourd'hui l'objet de travaux [20].

Un autre exemple d'utilisation des méthodes formelles est décrit dans [101]. Les auteurs présentent une étude de cas sur la vérification formelle d'un code industriel à Dassault Aviation. De plus, ils décrivent le lien entre l'application de ces techniques avec certains objectifs de certification dans [20].

Plus d'exemples sur l'utilisation des méthodes formelles dans l'industrie peuvent être trouvés dans [20], [21] et [92].

Premières observations chez Atos

Nous observons dans différents projets menés par Atos que quelle que soit la motivation ou le contexte pour l'utilisation des méthodes formelles, intégrer les méthodes formelles dans les phases de vérification nécessite le plus souvent l'intervention d'experts.

Nous observons plusieurs cas d'intervention possibles :

- les experts peuvent participer directement au projet dans le but de réaliser des activités à forte valeur ajoutée ;
- ou les experts peuvent être d'abord en charge de la formation des utilisateurs qui vont utiliser les techniques formelles et puis fournir un support à ces utilisateurs dans leurs travaux.

10. En anglais : Worst Case Execution Time (WCET).

Un mélange des deux cas n'est pas à exclure, puisqu'un projet de vérification logicielle peut être commencé par des experts et poursuivi par une équipe de développement avec le support d'experts.

3.3 Notre contexte technique

3.3.1 Le langage de modélisation

En Ingénierie Dirigée par les Modèles, Atos développe aujourd'hui une forte compétence sur l'utilisation industrielle du standard UML pour la modélisation de logiciels. Par exemple, Atos a décidé au début de cette thèse de s'appuyer sur le langage UML pour la phase de spécification logicielle d'un projet de système embarqué avionique pour le compte d'Airbus [50, 51]. Outre son utilisation à Atos, le standard UML présente également un fort intérêt de la part du monde industriel.

Utilisation d'UML dans l'industrie

Académiquement, les articles de recherche traitant d'UML sont très largement répandus. Industriellement, les témoignages de son usage sont principalement disponibles à partir d'enquêtes menées sur sa pratique auprès d'acteurs du monde logiciel. Ces enquêtes font état d'une utilisation conséquente dans l'industrie dans le cadre d'approche IDM. L'enquête menée par l'entreprise MediaDev [11] [61] en 2005 auprès de 500 développeurs européens a montré que plus de 97% des interrogés connaissaient UML et que 56% d'entre eux l'utilisaient dans leurs projets de développement logiciel. Une autre étude [60] a été réalisée en 2008 par des chercheurs de l'université d'Ottawa auprès d'une centaine d'acteurs du monde logiciel sur l'utilisation des approches centrées modèle par rapport aux approches centrées code dans les projets de développement. Cette étude a montré que le langage de modélisation le plus utilisé pour 52% des interrogés utilisant une approche centrée modèle était le langage UML.

En 2013, [105] a interrogé 50 ingénieurs logiciels issus de 50 entreprises différentes, et appartenant à plus d'une vingtaines de domaines différents, afin d'avoir un aperçu des applications industrielles d'UML et des raisons pour lesquelles UML ne serait pas utilisé industriellement. Parmi les interrogés, il apparaît que 30% d'entre eux utilisent UML. Cette utilisation est le plus souvent sélective : elle se limite à l'usage ponctuel d'un nombre restreint de diagrammes à certaines phases du développement. Pour les 70% restant, la majorité affirme connaître UML et l'avoir déjà utilisé mais aucune indication n'est donnée sur le fait qu'ils utilisent actuellement un autre langage de modélisation ou s'ils n'utilisent tout simplement pas de modèles dans leurs développements. L'enquête montre que cette non-utilisation d'UML est principalement due à des utilisations et des expériences antérieures sur quelques projets qui n'ont pas montré d'avantages significatifs par rapport aux pratiques classiques déjà utilisées. L'auteur précise cependant que les chiffres présentés dans cet article n'ont pas pour objectif d'être représentatifs de la

11. `http://www.mediadev.com/`

population globale des ingénieurs logiciel. L'enquête conclut que même si certains retours remettent en question l'étiquette de "*lingua franca*" donnée au langage UML, d'autres montrent qu'il existe industriellement des applications pratiques d'UML apportant des gains lors du développement, notamment en permettant aux utilisateurs de raisonner et de communiquer efficacement sur la conception logicielle.

Utilisation d'UML dans le domaine de l'embarqué

L'usage d'UML se retrouve également dans l'industrie du système embarqué. Différentes études ont été réalisées au cours des dernières années sur ce sujet. En 2004, [106] présente ces observations sur l'utilisation d'UML dans le cadre de projets de systèmes embarqués de grande échelle. Bien que l'une de ces observations soit que UML est souvent partiellement utilisé dans les projets étudiés, il observe aussi que l'utilisation de modèles UML améliore la communication entre les ingénieurs logiciels et systèmes ainsi que la communication avec le client final. UML offre notamment une meilleure visibilité et une plus grande confiance dans le système développé dès le début du cycle de développement. Plus récemment, [2] présente une étude sur les pratiques liées à UML et à l'IDM pour le développement de logiciels embarqués dans le cadre de l'industrie brésilienne. Cette étude regroupe les réponses de plus de 200 développeurs évoluant dans le domaine des logiciels embarqués et montre que 45% d'entre eux connaissent et utilisent UML dans leurs activités. Concernant les raisons de la non-utilisation d'UML ou de son utilisation partielle, seulement 0.9% des interrogés confirment utiliser un autre langage de modélisation. L'étude montre également que les diagrammes de classes, de cas d'études et de machines à états sont les diagrammes UML les plus utilisés.

3.3.2 La vérification formelle de code

Frama-C

Outre les compétences développées sur le standard UML dans le cadre de la modélisation logicielle, Atos possède également de fortes compétences sur l'application industrielle des méthodes formelles pour la vérification de code source. Une partie importante de ces compétences s'appuie sur l'utilisation de l'environnement Frama-C [12]. Frama-C se présente sous la forme d'une plate-forme modulaire libre et open-source dédiée à l'analyse formelle de programme C. Cette plate-forme s'appuie sur un langage d'annotations standardisé ACSL [13] pour la définition de propriétés et sur une palette de techniques formelles disponibles au travers de nombreux greffons. Ces techniques vont permettre d'analyser un programme, de vérifier l'absence d'erreur d'exécution ou bien de vérifier des propriétés sur un programme.

Par exemple, la technique d'analyse par valeur permet de s'assurer que le programme ne contient pas d'erreur d'exécution. Elle se base sur l'interprétation abstraite et sur

12. `frama-c.com`
13. ANSI/ISO C Specification Language.

le calcul des domaines de variation [14] des différentes variables d'un programme. Cette technique est implémentée dans le greffon Value Analysis [15] de Frama-C.

Un autre exemple de techniques disponibles est la technique de vérification par preuve déductive de code qui permet de vérifier des propriétés spécifiées sur un programme. Ce type de technique peut être appliqué grâce aux greffons WP [16] ou Jessie [17].

3.4 Synthèse

3.4.1 Premières observations

La problématique de cette thèse porte sur l'amélioration des processus de vérification logicielle en combinant les méthodes formelles et l'IDM. Comme nous l'avons vu dans la Section 3.2, les méthodes formelles sont déjà utilisées industriellement afin d'améliorer la qualité du logiciel produit. Cependant, nos observations à Atos montrent que leur introduction dans le cycle de développement et leur utilisation nécessitent l'intervention d'un expert.

Dans le cadre de l'amélioration des processus de vérification logicielle, nous pouvons alors nous demander s'il serait envisageable de tirer profit de l'IDM afin de pouvoir introduire l'utilisation des méthodes formelles auprès d'utilisateurs non experts en méthodes formelles, au sein de leur cycle de développement existant et en minimisant la dépendance aux experts. Cette proposition nous permettrait ainsi d'aboutir à une méthode à forte valeur ajoutée et accessible à un très grand nombre.

La prise en compte du savoir-faire existant est un facteur important lors de l'introduction de nouvelles méthodes auprès d'utilisateurs possédant leurs propres connaissances métier. Introduire une nouvelle méthode présentant une rupture avec les méthodes conventionnelles et sans tenir compte de ce savoir-faire est souvent synonyme de résistance au changement et, dans le pire des cas, de non-adoption de la méthode. Les travaux développés dans cette thèse s'intéressent plus particulièrement au contexte technique entourant une partie des activités de développement menées par Atos et décrit dans la Section 3.3. Dans ce contexte, nous savons notamment que l'utilisation du langage UML connaît une forte expansion dans les activités IDM d'Atos. Ce langage présente également l'avantage d'être connu et répandu auprès des acteurs du monde industriel comme académique. Nous savons également qu'Atos possède une expérience technique majeure dans l'utilisation de l'outil Frama-C pour la vérification logicielle et peut ainsi déjà proposer à ses clients des solutions pour l'application des méthodes formelles dans ce cadre.

A la lumière de ce contexte technique, nous pouvons aller plus loin dans notre réflexion et envisager de combiner la modélisation UML et l'outillage Frama-C afin de proposer une approche adaptée à des utilisateurs non experts et permettant l'amélioration de leur processus de vérification logicielle.

14. Les différentes valeurs possibles prises au cours de l'exécution.
15. `frama-c.com/value.html`
16. `frama-c.com/wp.html`
17. `krakatoa.lri.fr/jessie.html`

3.4.2 Définition d'un axe de recherche

Cette proposition d'approche représente une première direction à prendre pour nos travaux mais le standard UML est vaste et Frama-C propose de nombreuses techniques de vérification formelle. Pour démarrer nos travaux, nous avons besoin d'affiner cette proposition.

Nous avons vu que l'environnement Frama-C permet la vérification formelle d'un code source exprimé dans le langage C. Dans le cadre d'une preuve déductive, l'utilisateur doit seulement définir les propriétés à vérifier sous forme d'annotations ACSL et fournir le code source à prouver. Frama-C automatise ensuite le processus de preuve. Les propriétés définies sont des propriétés comportementales qui vont permettre de s'assurer que le comportement du code est conforme au comportement attendu. Dans un cycle de développement classique, le comportement attendu d'un logiciel est préalablement défini dans la conception dont est issu le code.

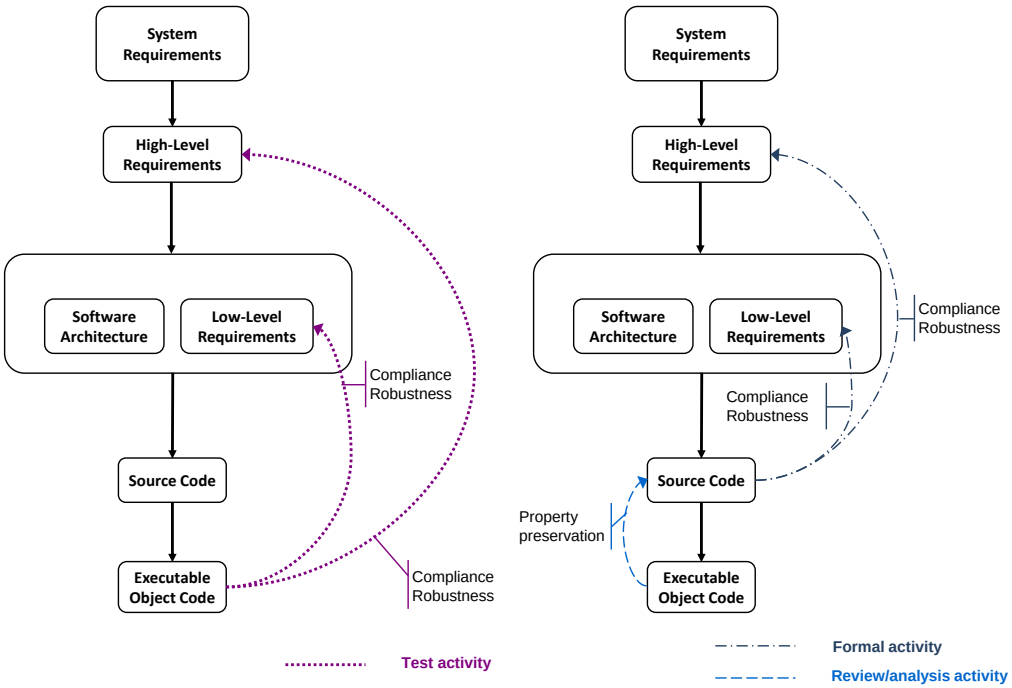
A partir de ce constat, nous pouvons définir un axe de recherche précis afin de répondre à notre problématique et en accord avec nos précédentes réflexions. Dans le but de proposer une approche adaptée à des utilisateurs non-experts et permettre l'amélioration de leur processus de vérification logicielle, nous allons chercher à automatiser la génération des propriétés comportementales sur le code à partir d'un modèle de conception UML afin d'automatiser entièrement, à l'aide de Frama-C, la vérification formelle de la conformité de ce code par rapport à son modèle de conception.

Cette vérification formelle s'inscrit dans les objectifs de certification définis pour le domaine avionique. Nous avons vu que l'extension DO-333 de la norme de certification DO-178 détaillait l'utilisation possible des méthodes formelles pour chaque phase du développement. Dans le cadre de la vérification de programmes, les objectifs de vérification alternatifs que cette extension définit en tenant compte de l'usage des méthodes formelles pour la vérification d'un programme par rapport à sa conception sont donnés Figure 3.2. La vérification de la conformité du code source par rapport à son modèle de conception y est définie comme pouvant être menée par des méthodes formelles. Notre axe de recherche est donc acceptable par rapport au contexte technique et à la norme de certification de notre environnement industriel.

3.4.3 L'approche proposée

Notre approche se focalise sur la génération automatique des propriétés à vérifier sur le code à partir du modèle de conception afin d'obtenir une étape de vérification formelle entièrement automatisée d'un point de vue de l'utilisateur. Ce principe a déjà fait l'objet d'une expérience préalable par Atos à partir d'un langage dédié d'automate défini par Airbus [41]. Notre approche s'inscrit dans la philosophie de ces travaux mais intervient dans un contexte technique différent puisque nous allons travailler avec le langage de modélisation UML et plus précisément avec les machines à états UML qui sont particulièrement adaptées à la représentation comportementale de logiciel. L'approche que nous proposons est visible Figure 3.3.

Elle s'intégre dans trois phases différentes d'un cycle de développement :



(a) Chemin classique avec les tests (DO-178)

(b) Chemin alternatif avec les méthodes formelles (DO-333)

FIGURE 3.2 – Chemin alternatif proposé pour la vérification de programmes

- la phase de conception qui permet la modélisation comportementale du logiciel à l'aide de machine à états UML ;
- la phase d'implémentation qui permet d'obtenir le code source. Le code est exprimé dans le langage C et peut être écrit manuellement ou généré automatiquement à partir du modèle de conception ;
- la phase de vérification qui comprend la génération automatique des propriétés comportementales à vérifier sous la forme d'annotations ACSL et la vérification formelle de ces propriétés sur le code par preuve déductive.

L'élaboration de l'approche est décrite dans les chapitres suivants. Le Chapitre 4 présente la définition du sous-ensemble UML utilisé et prescrit pour l'application de l'approche. Nous y décrivons les différentes étapes qui ont permis d'aboutir à un sous-ensemble UML formel pour nos travaux en s'appuyant sur les besoins industriels d'Atos. Le Chapitre 5 présente la génération des propriétés à partir d'un modèle de machines à états UML. Nous y décrivons la structure de l'implémentation d'une machine à états UML en C et les propriétés comportementales que nous générons. Enfin le Chapitre 6 décrit une implémentation de l'approche et son évaluation.

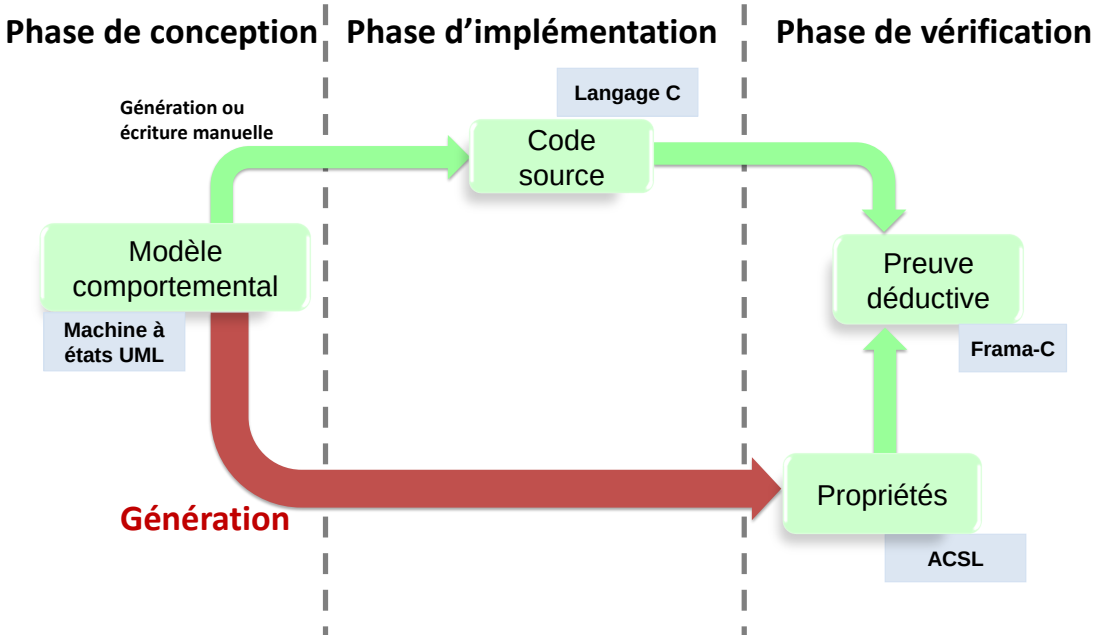


FIGURE 3.3 – Approche proposée

Chapitre 4

Définition d'un sous-ensemble UML pour la modélisation et la vérification formelle de logiciels embarqués

Afin de mener nos travaux, nous avons commencé par nous intéresser à un sous-ensemble UML sur lequel raisonner. Ce sous-ensemble doit pouvoir répondre aux besoins d'Atos, mais également être formel et proche du code. Nous nous sommes donc basé sur des travaux préléminaires que nous avons effectués dans le cadre d'un projet industriel au sein d'Atos. Ce projet nécessitait la définition d'un sous-ensemble UML pour la spécification logicielle d'un système avionique. Ces travaux ont donc été l'occasion de s'intéresser au standard UML et aux besoins industriels qui y sont liés. Un résumé de ces travaux concernant ce sous-ensemble est présenté dans la Section 4.1.

Cette utilisation industrielle d'une approche IDM étant pilote pour ce type de logiciel au sein d'Atos, nous avons recueilli les retours des utilisateurs afin d'obtenir un retour d'expérience sur la mise en place de cette approche. Un résumé de leurs retours et notre bilan sont donnés dans la Section 4.2.

Fort de cette expérience, nous nous sommes inspiré de ce sous-ensemble UML utilisé pour la spécification afin de définir notre sous-ensemble UML pour la conception logicielle, plus restreint mais plus proche du code. Afin de pouvoir l'utiliser dans la suite de nos travaux, ce sous-ensemble UML a fait l'objet d'une définition formelle de sa sémantique. Il est décrit dans la Section 4.3.

4.1 Travaux préliminaires : tentative de définition d'un sous-ensemble UML pour la spécification d'un logiciel embarqué

4.1.1 Le besoin industriel

Nos travaux portent sur la spécification d'un sous-ensemble logiciel d'un projet industriel de système avionique. Il doit permettre la gestion d'un groupe de composants logiciels et plus précisément des comportements qui leur sont liés. Il se base sur un composant "gestionnaire" qui va régir les autres composants. Le logiciel fonctionne par cycle, cadencé par un top d'horloge toutes les 30ms. A chaque cycle, le gestionnaire appelle séquentiellement chaque composant. Chaque composant ne réalise qu'une seule tâche par cycle : par tâche, nous entendons une série d'activités que le composant doit réaliser au cours d'un cycle. A la fin de chaque tâche, le composant se met en attente du prochain cycle et le gestionnaire autorise le composant suivant à réaliser la sienne. Si l'ensemble des composants n'a pas terminé ses tâches entre deux tops d'horloge, le logiciel est stoppé. Un exemple du comportement attendu du logiciel est disponible Figure 4.1.

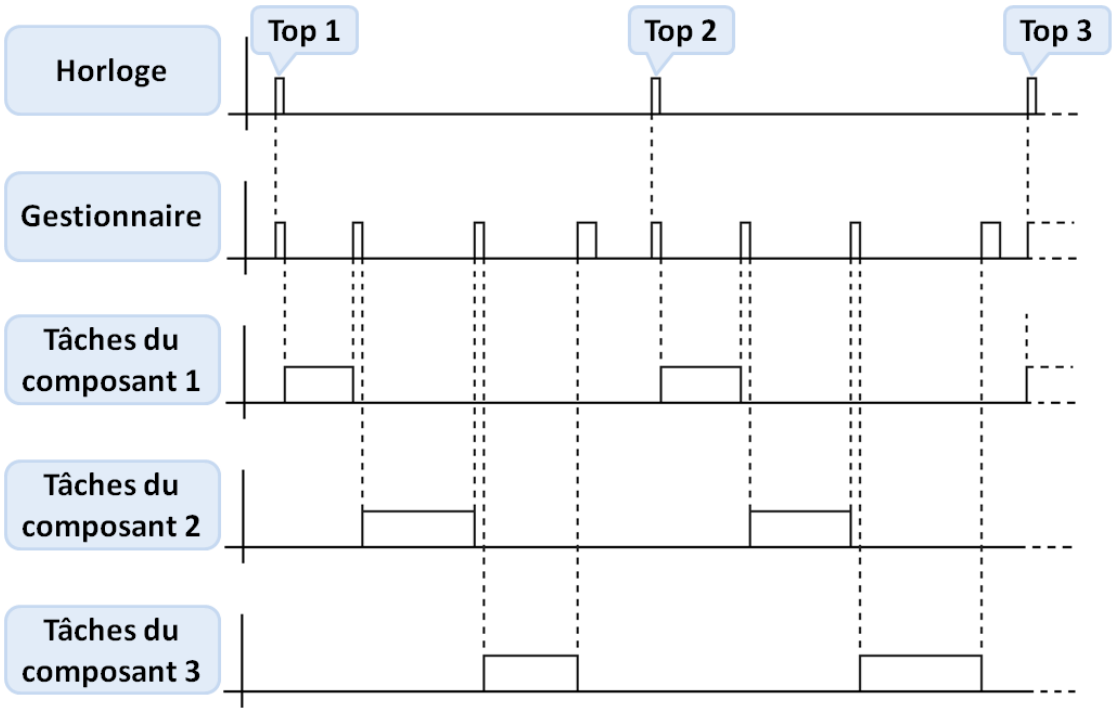


FIGURE 4.1 – Comportement global du sous-ensemble logiciel traité

La spécification de ce sous-ensemble logiciel a d'abord fait l'objet d'une spécification en langage naturel. Le sous-ensemble logiciel comptait environ 500 exigences au début du projet et leurs spécifications en langage naturel étaient de très grandes tailles et soumises

à interprétation. Il était difficile pour les équipes de traiter autant d'informations avec les ambiguïtés qui étaient soulevées à leur lecture. Cette complexité du sous-ensemble logiciel et ce premier travail de spécification a permis de souligner les limites de l'usage d'une méthode classique de spécification en langage naturel pour le cas de ce sous-ensemble logiciel, et a amorcé le choix de l'utilisation des modèles.

Des premiers diagrammes d'automates ont été produits pour spécifier les exigences. Ces diagrammes d'automates étaient exprimés en UML. Néanmoins, même si la sémantique était intuitive, elle n'avait pas été clairement définie pour les utilisateurs. La norme UML est vaste et les équipes chargées de la spécification n'utilisaient qu'une partie de la norme dans un contexte spécifique. Les aspects synchrones du système modélisé nécessitaient notamment une utilisation et un agencement particulier de certains concepts UML. Cette absence de définition précise des concepts nécessaires et de leur sémantique s'est rapidement fait ressentir et les équipes se sont donc tournées vers le pôle Méthodes & Outils d'Atos afin de définir le formalisme requis pour la spécification du sous-ensemble logiciel [1].

Ce choix de l'UML pour la spécification a donc été induit par les travaux initiaux précédemment décrits, mais également par l'expertise d'outillage disponible au sein d'Atos. Comme expliqué Section 2.1.6, la plate-forme open-source TOPCASED regroupe des outils pour l'édition de modèles UML mais également pour mener des activités d'analyse des exigences, de simulation de modèles, de tests, de génération de code et génération documentaire, etc. A l'époque du projet, Atos était responsable du développement et de la maintenance de la plate-forme, lui permettant ainsi de pouvoir proposer à ses équipes projet de disposer, dans les meilleures conditions, d'un éditeur UML mais également d'un ensemble d'outils personnalisés pour les aider à exploiter les modèles.

4.1.2 Le sous-ensemble UML défini

Les diagrammes utilisés

Nous avons proposé ([50], [51]) dans ce contexte d'utiliser un formalisme minimaliste qui s'appuie sur les concepts disponibles dans le standard UML. Il s'agit d'un sous-ensemble du langage qui se base sur les diagrammes de machine à états (voir Section 2.2.2) et sur les diagrammes d'activité pour la représentation comportementale du sous-ensemble logiciel. Les diagrammes d'activité sont des diagrammes permettant de représenter les activités pouvant être incorporées dans les machines à états et représentent des comportements de plus bas-niveau que ces dernières. Ils permettent notamment de représenter des séquences d'actions et le flot des données naviguant entre elles. Notre sous-ensemble inclut également le diagramme de bloc, équivalent au diagramme de classe UML. Ce diagramme est issu du langage SysML, sous-ensemble étendu du langage UML destiné à la spécification de systèmes. Ce diagramme est plus approprié à la représentation structurelle de composants systèmes que le diagramme de classe UML qui est

1. Ces travaux ont été amorcés antérieurement à la thèse, notamment par Raphaël Faudou, Stéphane Duprat et Jean-François Rolland. Le début de la thèse s'est inscrit dans la continuité de ces travaux.

surtout utilisé dans le cadre de la conception et de la programmation objet. Le diagramme de bloc nous permet de représenter la structure des différents composants du sous-ensemble logiciel traité.

La restriction des concepts

Pour chacun de ces diagrammes, nous utilisons un nombre restreint de concepts disponibles. Les concepts propres au diagramme de bloc sont restreints à l'usage des *Block*, des *Association* et des *Datatype*. Les *Block* permettent la représentation des composants logiciels. Les *Association* représentent les connexions entre les composants. Les *Datatype* permettent de représenter les différents types de données nécessaires à la modélisation. Ces concepts sont suffisants pour la représentation structurelle du logiciel.

Pour la représentation du comportement au travers des machines à états, nous nous appuyons sur les différents types d'état pouvant être représentés (*State*, *CompositeState*, *SubmachineState* et *FinalState*). Nous autorisons l'utilisation d'une partie des *Pseudostate*, comprenant l'*Initial pseudostate*, le *Choice pseudostate* et les *Entry/Exit Point* nécessaires pour la définition de *SubmachineState*. L'ensemble des concepts liés aux *Transition* sont également accessibles.

Les diagrammes d'activité sont limités à l'utilisation d'un nombre restreint d'actions : l'*OpaqueAction*, qui est l'action de base des diagrammes d'activités pour la représentation d'une action simple ; les actions *CallOperationAction* et *CallBehaviorAction* qui représentent respectivement l'appel d'opérations et l'appel d'autres activités de manière synchrone. Nous nous basons également sur l'utilisation d'un certain nombre de nœuds de données pour la gestion des flots de données : les *Output/InputPin* qui permettent de représenter les points d'entrée et de sortie des données dans les actions, l'*ActivityParameterNode* pour la représentation des points d'entrée et de sortie des données dans une activité ; le *DataStoreNode* pour le stockage de données ; et le *CentralBufferNode* pour la gestion des flots de données. Enfin, nous autorisons également l'ensemble des nœuds de contrôle, tels que l'*Initial node* qui représente le point de départ d'une activité et le *Decision node* qui permet de modéliser un choix entre plusieurs chemins vers des actions.

4.1.3 Des patrons de conception pour la modélisation

L'utilisation des concepts de ces diagrammes a fait l'objet d'une définition de patrons de conception afin de guider l'utilisateur dans la modélisation du sous-ensemble logiciel visé. L'ensemble des concepts utilisés pour la modélisation et les patrons mis en place ont été regroupés dans un guide utilisateur et mis à disposition des équipes projet. Nous précisons ici cinq des principaux patrons.

Patron de représentation d'un composant logiciel

Chaque composant logiciel est représenté statiquement par un bloc. A chaque bloc est associé une machine à états qui représente le comportement du composant. En UML,

nous dirons que la machine à états est définie en tant que *ClassifierBehaviour* du bloc. Chaque état d'une machine à états peut contenir un comportement plus détaillé qui correspond à une activité UML. Un exemple de représentation d'un composant est visible Figure 4.2. Cet exemple décrit la représentation du composant M1. Sur le diagramme de bloc, le composant est représenté par un bloc nommé `M1`. Le diagramme de machine à états représente son comportement. L'activité exécutée par l'état `Initialisation` est représentée par l'activité `init_standard` dont le comportement est décrit par un diagramme d'activité.

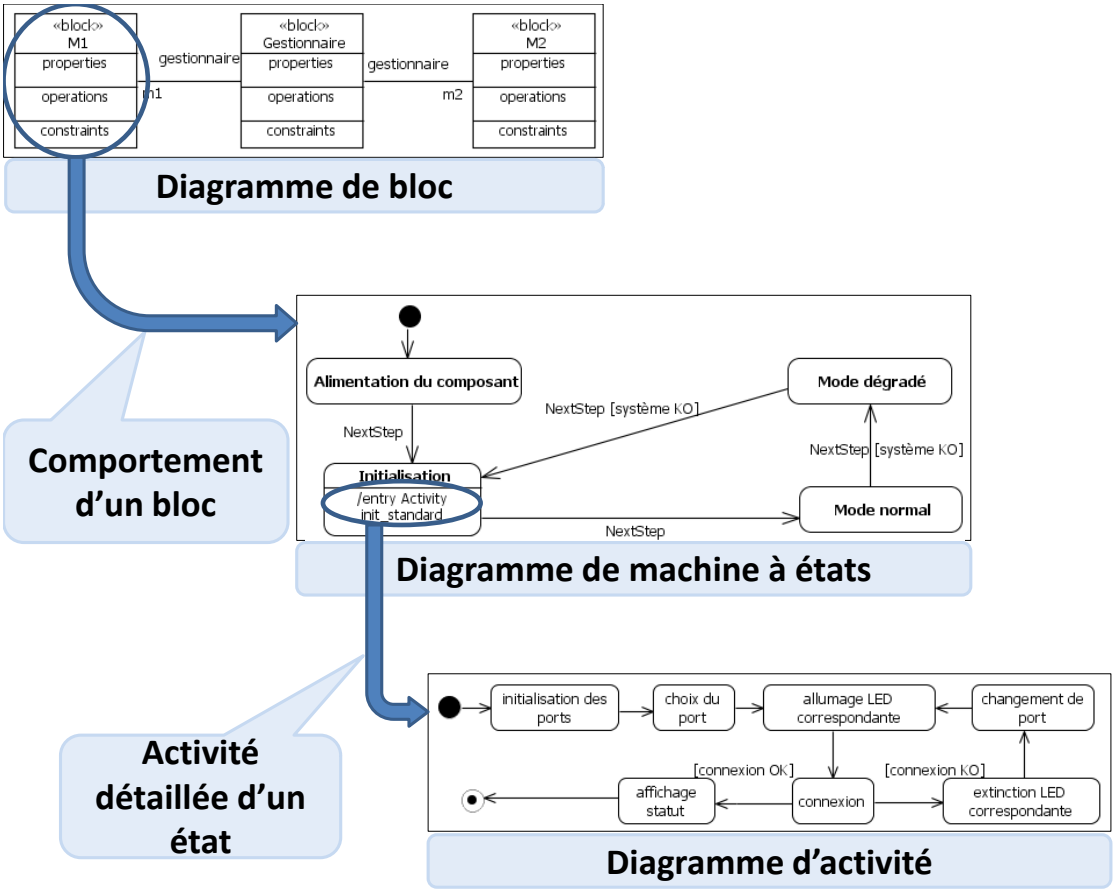


FIGURE 4.2 – Exemple de spécification d'un composant logiciel : représentation du bloc et de son comportement

Patron de représentation du fonctionnement synchrone et cyclique d'une machine à états

Nous souhaitons que chaque machine à états réalise une tâche par cycle. Une tâche est représentée par la succession d'un ou plusieurs états et par l'exécution des activités

qui leur sont liées. Pour représenter le fonctionnement cyclique d'une machine à états, il faut alors être capable de limiter l'évolution des machines à états tâche par tâche.

Pour chaque machine à états, nous définissons donc un événement correspondant au début d'un cycle. Il s'agit de l'unique événement extérieur pouvant être spécifié sur la machine à états. Cet événement, nommé *NextStep* dans notre sous-ensemble, est généré à chaque cycle. Pour chaque transition entre deux états représentant respectivement la fin d'une tâche et le début d'une autre, nous définissons un *Trigger* associé à cet événement. En complément, les transitions entre les états appartenant à une même tâche sont définies par l'événement généré par défaut à la fin des activités d'un état, le *completion event*. Ces transitions sont alors franchies automatiquement dès la fin des activités de l'état source, où à son entrée s'il ne possède aucune activité. Ainsi, lors de la génération d'un *NextStep*, la machine à états évolue automatiquement jusqu'à rencontrer une nouvelle transition franchissable uniquement à partir d'un événement *NextStep*, qui se produira au prochain cycle. Cette évolution tâche par tâche de la machine à états n'est cependant possible que si les états sont synchrones : il faut s'assurer que les activités d'un état se terminent avant le passage à un nouvel état. Ce synchronisme fait lui-même l'objet d'un patron de modélisation.

Enfin nous limitons la définition des *Effect* des transitions à une action de temps négligeable, dite ponctuelle, afin de garantir l'approche synchrone. Nous considérons qu'une activité est de temps négligeable si sa durée de traitement est négligeable par rapport à la durée d'un cycle. Ainsi, nous voulons nous assurer qu'une action lancée par une transition ne sera pas exécutée en parallèle avec l'activité d'un état.

Le patron pour la représentation cyclique des machines à états se base donc sur :

- la définition d'un événement unique nommé *NextStep* en tant que *Trigger* de toutes les transitions entre les états qui doivent être traités à des cycles différents ;
- la limitation des *Effect* des transitions à une action de temps négligeable ;
- la contrainte de définir uniquement des actions synchrones pour chaque état, garantissant ainsi le synchronisme de l'état.

Un exemple d'utilisation de ce patron est donné Figure 4.3. La transition sortante de l'état `Initialisation` possède un *Trigger* spécifié avec l'événement *NextStep* et un *Effect* qui est `Démarrer minuteur T1`. Cet *Effect* est ce que nous appelons une action de temps négligeable, elle est ponctuelle. Le déclenchement du franchissement de cette transition est donc engendré par l'occurrence de l'événement *NextStep* en début du cycle. Le franchissement de cette transition démarre le minuteur `T1` et déclenche le passage à l'état `Mise à jour.` La transition sortante de cet état possède également un *Trigger* spécifié avec l'événement *NextStep*. Une fois les actions de l'état `Mise à jour` terminées, la machine à états devra donc attendre la prochaine occurrence de l'événement *NextStep* au prochain cycle pour passer à l'état suivant. En revanche, la transition entre l'état `Pré-calcul` et l'état `Calcul global` ne possède pas de *Trigger*. Elle est donc automatiquement franchie au cours d'un même cycle.

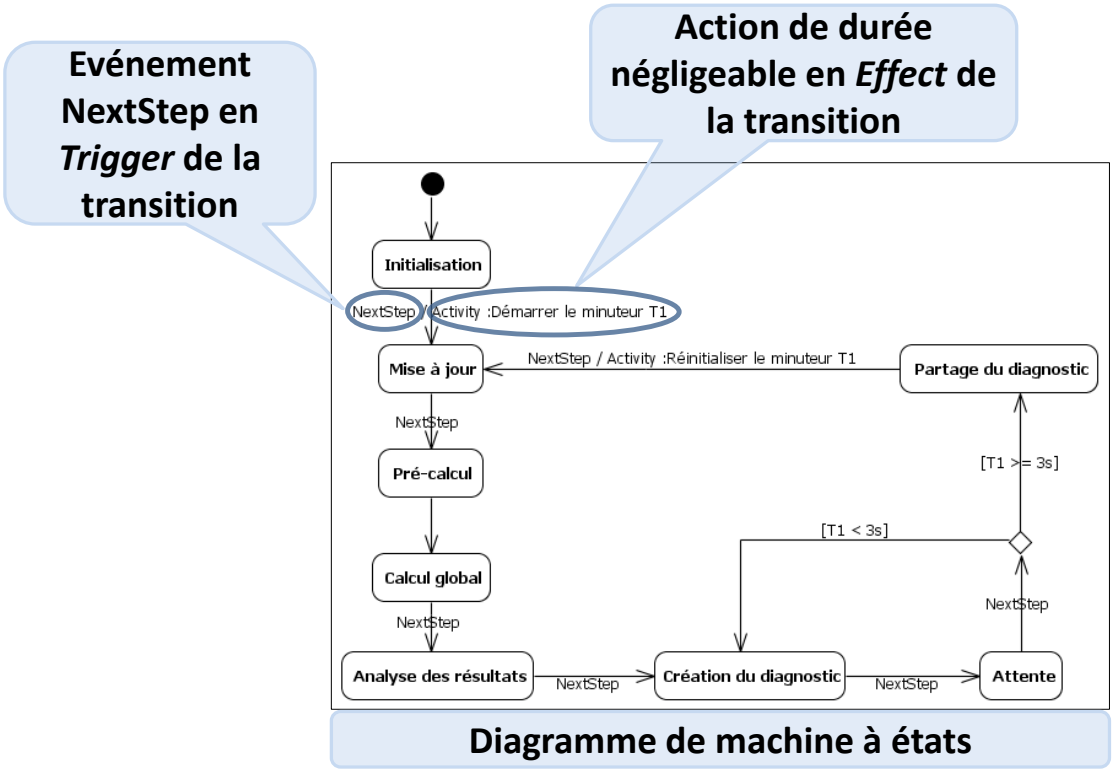


FIGURE 4.3 – Exemple de modélisation d'une machine à états synchrone

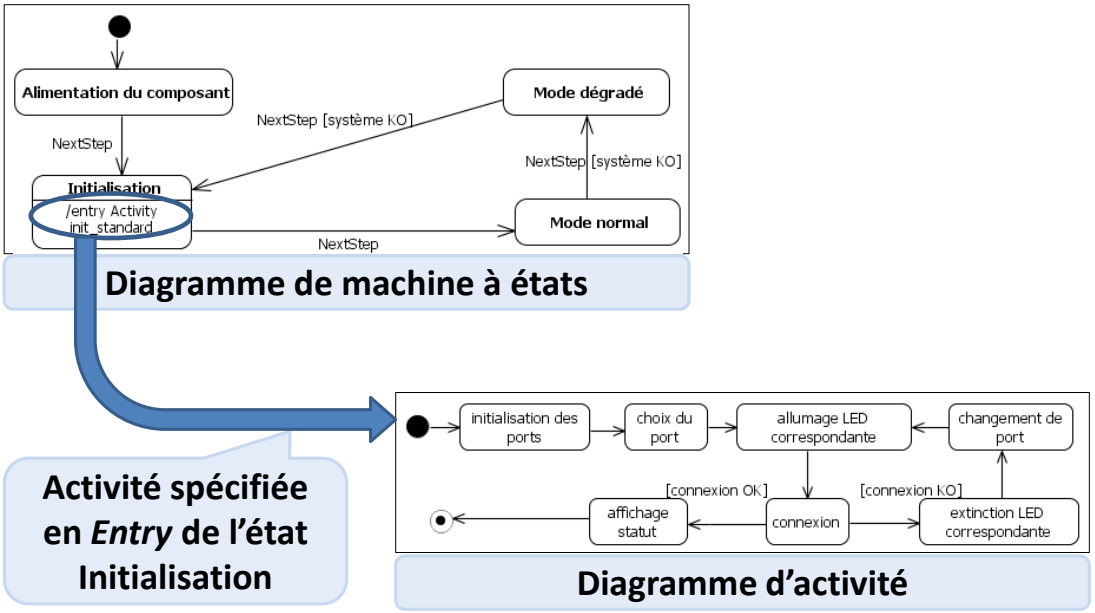


FIGURE 4.4 – Exemple de représentation du comportement d'un état

Patron pour le synchronisme d'un état

Afin de s'assurer du synchronisme des activités de chaque état, celles-ci sont définies uniquement en entrée [2] de l'état. Nous rappelons que l'avantage du traitement en entrée d'un état est d'être bloquant : l'état ne peut rien faire d'autre tant que l'activité lancée en entrée n'est pas terminée, garantissant ainsi son synchronisme. Une fois l'activité spécifiée en entrée terminée, l'état peut traiter les autres comportements qui lui sont attribués (dans le cas d'un *CompositeState* ou d'un *SubmachineState*) ou franchir une transition si cela est possible.

Le synchronisme des activités est également assuré par la limitation de notre sous-ensemble à l'utilisation d'éléments synchrones dans les activités, tels que les *OpaqueAction*, *CallOperationAction* et *CallBehaviorAction* décrites précédemment.

Un exemple est visible Figure 4.4. L'activité `init_standard` spécifiée en entrée de l'état `Initialisation` est représentée par un diagramme d'activité dont les actions s'exécutent et se terminent les unes après les autres dans un flot continu.

Patron pour le séquencement des traitements de chaque composant logiciel

Les différents composants du logiciel fonctionnent de manière séquencée à chaque cycle, comme décrit Figure 4.1. Ce séquencement est assuré par un composant particulier : le gestionnaire. Son comportement est représenté par une machine à états, qui va

2. En UML, on parle d'activité définie en *entry*.

appeler une activité à chaque cycle. C'est cette activité qui spécifie le séquencement de chacun des composants au cours du cycle.

Cette activité est composée d'une série de *CallOperationAction*, un pour chaque composant. Le *CallOperationAction* va permettre d'appeler une opération définie dans chaque bloc représentant un composant. L'appel de cette opération va uniquement déclencher la génération d'un événement de type *CallEvent*. Il s'agit de l'événement *NextStep* qui va indiquer le début d'un cycle à la machine à états et qui va donc lui permettre d'entreprendre ce que nous avons appelé une tâche. En UML, le *CallOperationAction*, si défini synchrone, est bloquant : il ne peut se terminer que lorsque le comportement déclenché par l'appel de l'opération est terminé (ici, ce comportement correspond à la tâche de la machine à états engendrée par l'événement *NextStep* généré par l'appel de l'opération). La machine à états d'un bloc (ou composant) ne peut donc pas évoluer tant que le traitement déclenché par l'appel de l'opération du précédent bloc n'est pas terminé. Cette limitation garantit ainsi le séquencement des différents composants.

Le patron pour le séquencement des traitements de chaque composant logiciel se base donc sur :

- la définition d'une opération pour chaque bloc représentant un composant logiciel ;
- une activité composée du séquencement attendu de *CallOperationAction* synchrone, chacun appelant l'opération d'un bloc ;
- le lien entre l'opération de chaque bloc et l'événement *NextStep* de la machine à états qui lui est associé.

Un exemple de séquencement est disponible Figure 4.5. Le diagramme de bloc indique que le `Gestionnaire` gère deux composants logiciel : `M1` et `M2`. Pour chacun de ces blocs, une opération est définie : `M1_op` pour `M1` et `M2_op` pour `M2`. Le comportement du `Gestionnaire` est tout d'abord décrit par un diagramme de machine à états. Dans cette machine à états, l'état `Séquencement` est atteint à chaque cycle. Cet état contient une activité nommée `séquencement_des_composants` qui est alors appelée à chaque cycle. Cette activité est représentée par un diagramme d'activité et est constituée d'une séquence de deux *CallOperationAction*, `appel M1_op` et `appel M2_op` qui font chacun respectivement référence aux opérations `M1_op` et `M2_op` définies dans le diagramme de bloc. Ces deux *CallOperationAction* vont chacun permettre d'appeler leurs opérations associées et ainsi permettre le séquencement des machines à états qui leur sont respectivement liées.

Patron pour la factorisation des comportements

La modélisation des spécifications de notre sous-ensemble logiciel se base sur une approche modulaire. Cette approche présente des avantages : elle donne la possibilité de factoriser des comportements, les utilisateurs pouvant spécifier et réutiliser des comportements élémentaires pour l'élaboration des composants logiciels. Cette approche limite également la taille de chaque machine à états en exposant un nombre réduit d'états rendant la lecture des modèles plus aisée.

La factorisation des comportements se base sur la notion de *SubmachineState* décrite en Section 2.2.2 et qui permet, à des machines à états, de faire référence à des compor-

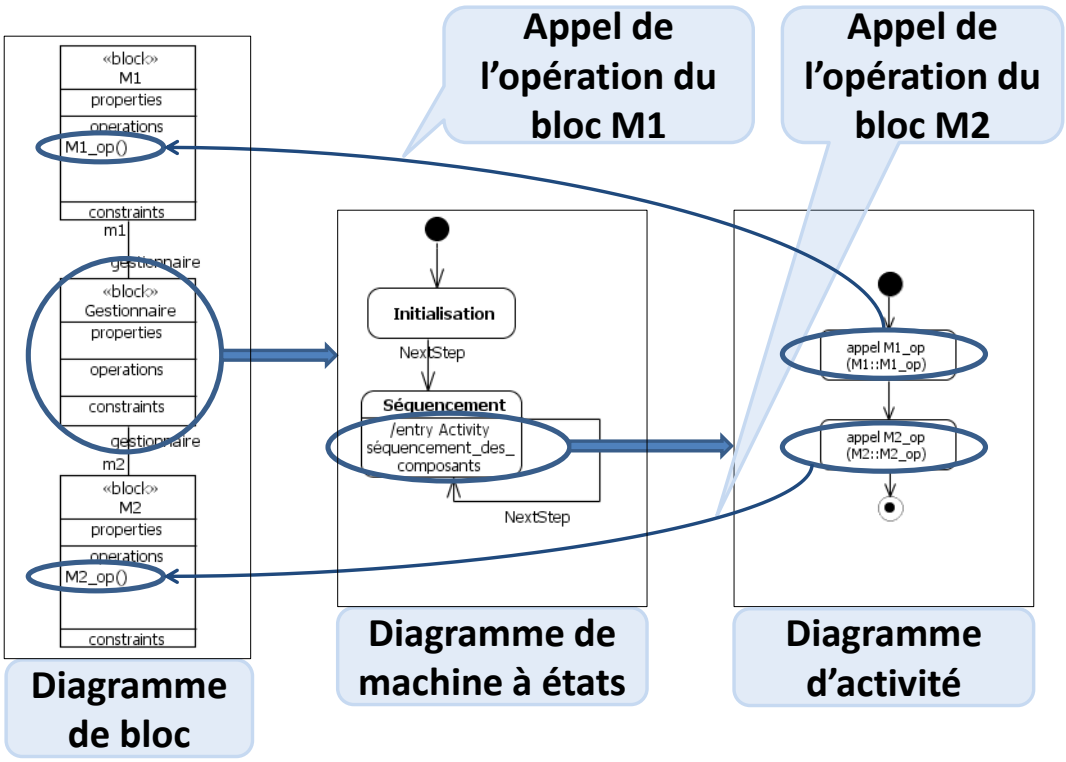


FIGURE 4.5 – Exemple d'appel séquencé des composants

tements définis dans d'autres machines à états. Dans notre sous-ensemble, les liens entre les comportements sont représentés sur le diagramme de bloc par des associations entre blocs. Chaque comportement utilisé dans un *SubmachineState* est représenté par un bloc qui est lié à tous les blocs faisant appel à ce comportement dans leur machine à états. Cette factorisation peut également être mise en place au niveau des activités avec l'utilisation d'un *CallBehaviorAction* qui permet l'appel d'activités dans une activité.

Un exemple de factorisation est représenté Figure 4.6. La machine à états 1 fait appel à la machine à états 2 (`Mode dégradé`) à partir de l'état `Mode dégradé`, qui elle-même fait appel à la machine à états 3 (`Demande diagnostic`), plus basique, dans l'état `Demande diagnostic`. Notons que la machine à états la plus élémentaire ne possède plus de *SubmachineState*. Nous remarquons sur chaque machine à états, la définition de points de connexion d'entrée (tels que `démarrage`) et de points de connexion de sorties (tels que `ko`) qui se retrouvent sous forme de pseudo-états portant le même nom dans les machines à états référencées. Par exemple, dans la machine à états `Mode dégradé` (au centre), `démarrage` est un pseudo-état de type *EntryPoint* et `ok` et `ko` sont des pseudo-états de type *ExitPoint*.

4.2 Bilan de l'utilisation de la modélisation pour la spécification

Dans le contexte industriel entourant Atos, l'utilisation d'un formalisme UML pour la spécification logicielle dans le cadre d'un projet de système embarqué avionique n'a jamais été mise en place auparavant. Cette première mise en œuvre a donc été considérée comme "pilote" . Par conséquent, il était nécessaire de capitaliser le retour d'expérience qui en émane et plus précisément le ressenti des utilisateurs. Les retours qui ont été récoltés portent sur l'utilisation et l'impact des modèles dans le contexte du sous-ensemble logiciel visé. Ils ne portent pas précisément sur le formalisme de modélisation mis en place. Ces retours ont été capitalisés au cours d'entretiens, allant de 30 min à 1h30 réalisés auprès de quatre membres du projet intervenant sur différentes phases du cycle de développement. Ils avaient une connaissance préalable, forte (formation théorique et mise en pratique sur des cas industriels) ou sommaire (formation théorique), de l'IDM et du langage UML et possédaient déjà une expérience dans le domaine des projets de systèmes embarqués aéronautiques. C'est un résumé de leurs retours qui est présenté dans les paragraphes suivants.

4.2.1 Les retours utilisateurs

Les apports de la modélisation

La modélisation a tout d'abord apporté un certain nombre d'avantages aux équipes projet. L'un des principaux retours des utilisateurs porte sur l'apport visuel de la modélisation : selon eux, elle a permis d'obtenir une plus grande clarté sur la spécification. C'était le but premier attendu : face au nombre important d'exigences à modéliser, il

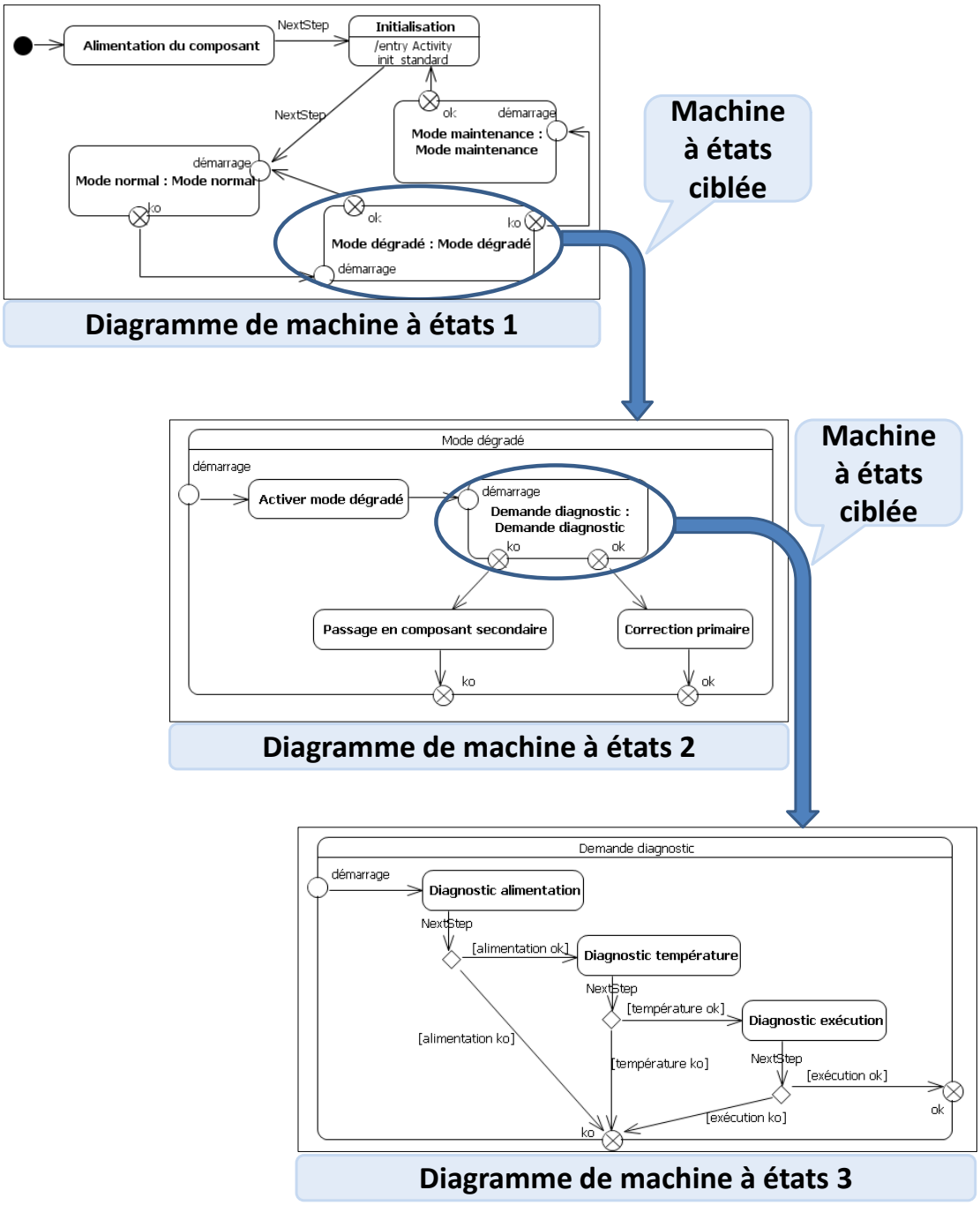


FIGURE 4.6 – Exemple d'encapsulation de machine à états à l'aide de SubmachineState

était nécessaire de trouver un moyen de maîtriser leur complexité. D'après les retours, les modèles ont rempli cette tâche et ont rendu les spécifications plus descriptives que celles réalisées en langage naturel, et non soumises à interprétation. Le gain en lisibilité est très important pour les utilisateurs et la possibilité de pouvoir naviguer facilement dans le modèle grâce au *"support numérique"* représente un avantage non négligeable. Certaines réserves ont néanmoins étaient émises : le contexte du sous-ensemble logiciel traité est particulier au vu du grand nombre d'exigences à respecter, et certains des utilisateurs interrogés ne sont pas sûrs que ce gain au niveau descriptif aurait été aussi important pour un sous-ensemble logiciel plus simple.

Les utilisateurs ont également exprimé le fait que ce gain en clarté et en description a rendu la prise en main de la spécification plus intuitive et plus facile qu'avec les spécifications en langage naturel précédemment utilisées. Une des équipes projet a réalisé un test informel en interne, en donnant les spécifications dites *"papiers"* et les spécifications sous forme de modèles à deux personnes arrivant sur le projet. Les modèles ont clairement été plus appréciés par les utilisateurs. Ce gain de prise en main leur a permis notamment de faire intervenir de nouveaux acteurs sur le projet beaucoup plus rapidement qu'auparavant.

Un autre retour positif porte sur l'apport au niveau de la communication et de la coordination au sein du projet. Les utilisateurs ont trouvé qu'il était plus facile de communiquer entre acteurs du projet à l'aide du modèle, comme par exemple entre les personnes chargées de la spécification et les concepteurs. Ils ont également noté qu'il était possible de mieux se coordonner et de mieux répartir les différentes tâches à effectuer sur le projet : les modèles permettent de pouvoir s'approprier et comprendre une partie des spécifications sans pour autant avoir besoin de connaître son ensemble et permettent ainsi à différents utilisateurs de travailler sur des parties restreintes de la spécification, à moindre coût d'investissement.

L'apport de la modélisation pour d'autres phases du projet a également été souligné. Les concepteurs ont été à même de pouvoir discerner manuellement des patrons de code à partir des modèles de la spécification. De plus, pendant la phase de tests, l'organisation de la modélisation a permis de rendre les tests plus unitaires et mettre en avant des stratégies de tests plus performantes.

L'apport d'une certaine cohérence a également été apprécié pour la rédaction documentaire. Cette idée de *"ne rien oublier"* lors du passage des modèles aux documents de livraison, grâce à la génération automatique de documents, a été exprimée par un utilisateur. La disponibilité de nombreux outils dans TOPCASED pour répondre aux besoins des équipes projet a également été perçue par certains interrogés comme un gain en adaptabilité. Les utilisateurs ont notamment pu utiliser l'outil TOPCASED Requirement [44] pour la traçabilité des exigences et l'outil GenDoc2 [3] pour la génération documentaire.

3. `http://marketplace.eclipse.org/content/gendoc2`, greffon Eclipse pour la génération de documents à partir de modèle.

Les limites de l'emploi de la modélisation

Même si l'usage de la modélisation a été très bénéfique pour le projet, les utilisateurs ont néanmoins relevé certains désavantages à son utilisation.

L'une des principales limites rapportées par les utilisateurs est l'étendue du langage de modélisation utilisé. Les possibilités induites par le standard UML sont nombreuses (plus d'une dizaine de types de diagramme différents et quelques centaines de concepts à manipuler) et les notions exprimées, même dans le sous-ensemble limité, ont parfois été sujettes à interprétation. Les utilisateurs ont émis des hypothèses sur ces problèmes d'interprétation. Certains l'expliquent par le fait que lorsque l'utilisateur aborde la modélisation logicielle, il pense avoir une connaissance complète du langage UML grâce à sa formation académique, formation présentant le plus souvent un aperçu d'UML via la Conception Orientée Objet et les aspects asynchrones. Mais ces concepts sont soumis à restriction dans notre sous-ensemble et il y a parfois méprise sur l'utilisation de certains d'entre eux lors de la modélisation. D'autres pensent que réussir à mettre en place l'usage d'une telle modélisation était trop complexe pour réussir dès la première tentative.

Un des utilisateurs a également exprimé le fait que les concepts décrits dans le guide n'était pas assez clairs pour une personne connaissant peu UML et que le guide avait été écrit plus comme une norme que comme un guide pour l'utilisateur. L'un des principaux problèmes qui a découlé de ces problèmes d'interprétation est la création d'un grand nombre de duplicata de code lors d'une première phase d'implémentation. Cependant, les utilisateurs ont observé de nettes améliorations après une revue et une nouvelle appropriation du sous-ensemble UML utilisé. Il y a notamment eu un gain au niveau de la qualité de la conception, qui était plus mature et plus simple que la version précédente.

Une autre difficulté qui a été signalée est l'aspect expérimental de l'approche mise en place : elle n'avait encore jamais été utilisée par les équipes Atos pour ce type de projet, et la définition de notre sous-ensemble UML et le choix des outils pour la modélisation ont eu lieu en parallèle du projet. Les utilisateurs ont donc parfois dû faire face à des difficultés induites par cet aspect. A titre d'exemple, les problèmes d'interprétation décrits précédemment sont en partie dus à cet aspect, étant donné que le sous-ensemble UML utilisé a évolué au cours du projet.

Une autre limite rencontrée par les utilisateurs concerne l'outillage. Un retour utilisateur fait état d'un trop grand nombre d'informations graphiques sur les outils utilisés. Même si nous utilisons un sous-ensemble du langage UML pour la modélisation, l'éditeur TOPCASED qui a été utilisé est l'éditeur classique qui couvre l'ensemble du standard. Il y a donc des représentations qui ne sont pas utiles à notre modélisation et c'est à l'utilisateur de savoir ce qu'il doit ou non utiliser pour la représentation.

Enfin une dernière difficulté concerne le fait que le formalisme UML n'est pas utilisé sur l'ensemble du cycle du projet. La modélisation UML n'intervient que pour la phase de spécification pour spécifier les exigences qui vont être utilisées tout au long du projet. Même si cette modélisation fournit des apports pour d'autres phases, la phase de conception est réalisée à partir d'un langage et d'un outil propre au client. Le fait de devoir passer manuellement d'un formalisme à un autre a été perçu comme quelque peu déroutant et frustrant par les utilisateurs.

Les préconisations des utilisateurs pour un emploi futur

Hormis les apports et les limitations de l'emploi de la modélisation, nous avons également récolté l'avis des utilisateurs sur leur vision idéale du projet, afin de capitaliser les bonnes conduites à adopter pour de futurs projets.

Selon les utilisateurs, l'amélioration principale à mettre en place est de pouvoir utiliser la modélisation amont sur tout le cycle de développement du projet, de la phase de spécification à la phase de tests. Nous avons fait remarquer précédemment qu'il y avait un changement d'outil et de formalisme entre la phase de spécification et la phase de conception, changement qui était fastidieux pour l'utilisateur. Une seconde amélioration proposée par les utilisateurs serait de pouvoir tester le modèle en amont, par vérification de modèle ou par simulation, afin de s'assurer de la correction de ce qui est modélisé et ainsi gagner en fiabilité et gagner du temps en maintenance.

Enfin, une bonne conduite à adopter serait de définir dès le début du projet le formalisme et l'environnement de travail à utiliser. Les retours ont montré que l'aspect expérimental de l'approche avait été source de nombreuses difficultés. L'un des utilisateurs a également exprimé le fait qu'il était nécessaire, pour un futur emploi, de s'assurer que tout le monde ait la même interprétation des concepts, afin d'éviter des erreurs de modélisation aval.

4.2.2 Synthèse

Le sous-ensemble logiciel du projet qui a été traité est de grande taille et soumis à de fortes contraintes de développement issues aussi bien du domaine aéronautique que des processus industriels qu'il doit respecter. Dans notre contexte, son développement est apparu comme pilote pour la mise en place d'une modélisation UML en phase de spécification. Les retours des utilisateurs sont encourageants et suggèrent que l'approche IDM pourrait être efficace pour traiter des projets similaires. En dépit des difficultés auxquelles ils ont dû faire face au cours du développement, ils sont convaincus que l'approche a été bénéfique pour le projet. Les gains sur l'écriture, la compréhension et la communication des spécifications ont clairement été exprimés et le support de l'outil TOPCASED a permis d'accompagner les équipes projet et de les soutenir durant les différentes phases du projet en leur fournissant des outils capables de répondre à leurs besoins.

Les retours utilisateurs sur les difficultés rencontrées permettent également d'avoir un aperçu des problèmes à corriger sur l'approche mise en place. Les problèmes d'interprétation rencontrés lors de l'usage du sous-ensemble UML représente la principale difficulté rapportée. Outre les hypothèses supposées par les utilisateurs sur ces problèmes, nous pensons qu'ils pourraient être dus à une stratégie de communication inadéquate du guide utilisateur auprès des équipes projet ou bien à une rédaction initiale de ce guide mal adaptée au public ciblé. A l'avenir, il faudrait par exemple mettre en place des formations amont pour s'assurer de l'interprétation de la sémantique des concepts utilisés, ou bien une rédaction des guides pour la modélisation plus adaptée à l'utilisateur. D'autres limites rapportées ont d'ores et déjà étaient corrigées, telles que le problème qui concer-

nait l'outillage et les problèmes d'adaptation de l'éditeur TOPCASED au sous-ensemble UML ciblé. Les dernières versions de TOPCASED permettent désormais de personnaliser la palette de l'éditeur suivant ses besoins et ainsi limiter les concepts disponibles pour la modélisation.

Les préconisations envisagées par les utilisateurs dans le cadre d'une future utilisation de l'approche tendent vers la mise en place d'un cycle de développement basé sur un unique formalisme de modélisation et qui posséderait des étapes de vérification dès les phases amont du développement. Comme nous l'avons abordé dans la Section 2.1.5, la vérification de modèles est un sujet depuis longtemps traité et de nombreux outils existent aujourd'hui tels que par exemple TINA [16] ou OBP [37]. Nous avons nous-même déjà proposé un processus pour la vérification de modèles de spécification dans [34, 36, 35] , où nous soulignions la nécessité de modéliser *"juste"*. Ces travaux presentent une vérification de la spécification, par rapport à des propriétés exprimées en OCL, à partir de model-checkers existants tels que UMLtoCSP [24] ou UML2Alloy [5]. Ces travaux introduisent également une aide à la conception de modèles basée sur une génération automatique et optimale d'une partie de ces modèles à l'aide de propriétés OCL et de solveurs de contraintes basés sur des techniques SAT.

Le cycle de développement suggéré par les préconisations des utilisateurs s'intègre parfaitement avec l'approche envisagée Figure 3.3 pour répondre à la problématique traitée dans cette thèse. Il serait alors possible d'imaginer un cycle de développement basé modèle intégrant des phases de vérification automatique dès la spécification et jusqu'à l'implémentation, comme nous l'avons suggéré dans [52, 49]. Une vision idéale de ce cycle est représentée Figure 4.7. Il intègre différentes étapes de vérification à chaque phase de développement de la branche descendante du cycle en V. De plus, il préconise l'utilisation du formalisme UML pour les phases de spécification et de conception détaillée avec différents niveaux de raffinements, permettant ainsi de conserver le même formalisme de modélisation au cours du cycle de développement.

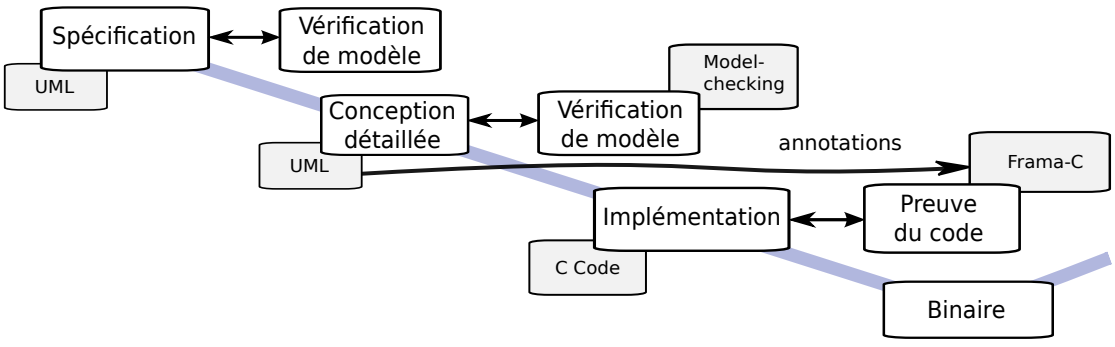


FIGURE 4.7 – Cycle de développement idéal

4.3 Définition de notre sous-ensemble UML pour la conception

Le sous-ensemble UML qui a été décrit précédemment intervient durant la phase de spécification d'un sous-ensemble logiciel. Les travaux dont il a fait l'objet présentent l'avantage de mettre en évidence le type de modélisation attendu et pouvant être rencontré au cours du développement logiciel d'un système embarqué dans le cadre des activités menées par Atos. Cependant, ce sous-ensemble UML demeure encore trop informel pour conduire nos travaux sur la vérification de code à partir de méthodes formelles. Par exemple, certaines descriptions, telles que le comportement représenté dans les activités ou le contenu des gardes des transitions des machines à états, sont encore exprimées en langage naturel. La poursuite de nos travaux nécessite un sous-ensemble UML intervenant durant la phase de conception détaillée qui est plus proche du code et qui doit être strictement formel.

Nous nous sommes donc inspirés de notre sous-ensemble UML pour la spécification logicielle pour en déduire un sous-ensemble UML dédié à la conception détaillée. Dans le cadre de nos travaux, ce sous-ensemble est limité à la représentation comportementale d'un logiciel à l'aide de machines à états. Comme expliqué Section 2.2, l'utilisation de machine à états est courante pour ce type de représentation. Nous intéresser spécifiquement aux machines à états nous permet alors de toucher une communauté plus large avec nos travaux, tout en répondant à des problèmes traités par Atos.

4.3.1 Description

Dans notre sous-ensemble, nous considérons qu'une machine à états est pilotée par une horloge et peut réaliser un certain nombre d'activités entre chaque top d'horloge, similairement aux machines à états précédemment décrites. Chaque top d'horloge marque le début d'un cycle de la machine à états. Nos machines à états sont composées d'états simples dans lesquels des activités peuvent être exécutées, ces activités restant définies pour une exécution synchrone à l'entrée de l'état. Les transitions de nos machines à états peuvent être composées d'un *Trigger* et d'une *Guard*. Cependant, une transition est désormais sans effet de bord i.e. elle ne porte aucune action ou activité définie dans son *Effect*. Le *Trigger* peut être associé à deux événements possibles dans notre sous-ensemble : l'événement *tick* qui représente un top d'horloge et l'événement *completion event* qui est l'événement par défaut défini dans UML. La *Guard* est une expression booléenne, une formule de la logique du premier ordre avec des variables, des constantes et sans quantificateur, exprimée dans un sous-ensemble du standard OCL [4]. Les seuls symboles de prédicats que nous acceptons dans une *Guard* sont les opérateurs de comparaison arithmétique : $<, >, \leq, \geq, =, \neq$. "$x =< 3$ *and* $y = true$" est un exemple de *Guard* pouvant être défini, où x et y sont deux variables de notre système. Le fonctionnement d'une transition reste inchangé : une transition est déclenchée à l'arrivée de son événement défini en *Trigger* et est franchie à condition que sa *Guard* soit vraie. Enfin, nous

4. Object Constraint Language, `http://www.omg.org/spec/OCL/`

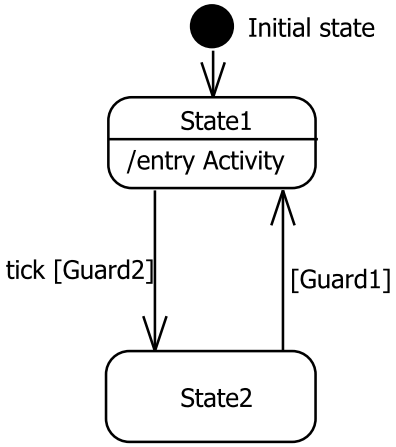


FIGURE 4.8 – Représentation des éléments disponibles dans le sous-ensemble

autorisons l'usage d'un unique pseudo état : l'état initial.

Nous rajoutons deux hypothèses à notre sous-ensemble. La première est que toutes les activités qui commencent après un top d'horloge se terminent avant le prochain top d'horloge. Comme nous ne sommes pas intéressés par les propriétés temporelles du système, nous admettons donc l'hypothèse de synchronisme définie dans [15] : toute réaction du système est instantanée. La deuxième hypothèse est que nous ne considérons que des machines déterministes i.e. nous n'autorisons pas les transitions concurrentes.

Un exemple purement descriptif des éléments constituant notre sous-ensemble est disponible Figure 4.8. Cette machine à états est constituée d'un état initial `Initial state` et deux états `State1` et `State2`. L'état `State1` possède une activité définie en entrée. La transition de `State1` à `State2` possède un *Trigger* associé à l'événement *tick* et une *Guard* `Guard2`. La transition de `State2` à `State1` possède uniquement une *Guard* `Guard1`. Nous rappelons que si une transition n'est étiquetée par aucun événement alors l'événement qui la déclenche est implicitement le *completion event*.

Notons que les activités décrites dans les états sont un sous-ensemble des diagrammes d'activités UML. L'étude de ces activités et leur lien avec la vérification ont fait l'objet de travaux avancés au sein d'Atos hors de cette thèse. Ce sous-ensemble ne s'inscrit pas réellement dans les travaux de cette thèse mais, à titre d'information, des premiers résultats ont été publiés dans [42] et une description du sous-ensemble des activités UML envisagé est disponible dans l'Annexe A.

4.3.2 Définition formelle

Notre sous-ensemble est issu de la version 2.4.1 du standard UML, limité aux machines à états. La sémantique actuelle d'UML est semi-formelle. Elle s'appuie sur une association de méta-modèles, de contraintes OCL et de descriptions en langage naturel.

Des propriétés de certains concepts sont également soumises à, ce qui est appelé dans le standard, des points de variation sémantique [5] qui laissent toute liberté à l'utilisateur de définir l'interprétation de ces propriétés. Dans le but d'éviter toute ambiguïté et de pouvoir utiliser rigoureusement les méthodes formelles, nous avons besoin de formaliser la sémantique du sous-ensemble UML que nous utilisons. Dans le cas particulier des machines à états UML, de nombreux travaux existent sur la formalisation de leur sémantique. Les travaux de [31] donne un aperçu de l'état de l'art actuel. Les auteurs recensent 26 approches réparties en trois catégories. Tout d'abord, les travaux exprimant la sémantique à partir de notations et de concepts mathématiques. Par exemple, [111] utilise des Systèmes de Transitions Labélisés (LTS) exprimés dans un langage de spécification algébrique pour décrire la sémantique ; ou bien [19], qui fait usage des machines à états abstraits (ASM). Une seconde catégorie regroupe les approches exprimant la sémantique comme un ensemble de règles de réécriture. Par exemple, les travaux de [129] et [65] utilisent des transformations de graphes. Enfin la dernière catégorie concerne les approches basées sur la traduction des machines à états UML dans un autre langage formel comme, par exemple, [6] qui définit une sémantique en PVS (Prototype Verification System) et [87] propose une sémantique implémentée en PROMELA en vue du model checking. Aucune de ces approches ne prend en compte tous les concepts des machines à états UML. Notre approche peut être classée dans la première catégorie : nous définissons une sémantique mathématique très simple, limitée aux concepts utiles dans notre contexte. L'avantage de définir notre propre sémantique est de pouvoir disposer d'une sémantique simple et maîtrisée, qui sera plus facile à étendre et dont la définition est beaucoup moins coûteuse que l'adaptation d'une sémantique existante à notre sous-ensemble.

La sémantique que nous proposons ([54, 55]) reste entièrement compatible avec la sémantique du standard UML bien qu'elle introduise de nouveaux concepts propres à notre sous-ensemble. De plus, afin d'avoir une sémantique aussi simple que possible, nous faisons abstraction de la notion d'événement et nous utilisons seulement le concept de *Run-to-completion step* défini en UML, et décrit dans la Section 2.2.2, que nous allons spécialiser pour chacun de nos deux événements.

Syntaxe

Nous définissons une machine à états comme un $6-$tuplet $\langle S, s_0, V, v_0, \mathit{Cycle}, \mathit{Cycle}_0 \rangle$, avec :

- S l'ensemble des états de la machine à états ;
- $s_0 \in S$ l'état initial ;
- V l'ensemble des fonctions de valuation qui associent à une variable de l'ensemble des variables accessibles VAR de la machine à états, un élément du domaine du discours et v_0 la fonction de valuation pour s_0 ;
- $\mathit{Cycle} : S \times V \rightarrow S \times V$ la fonction qui, à chaque top d'horloge (*tick*), va retourner le nouvel état courant et la nouvelle fonction de valuation de la machine à états

5. En anglais : Semantic Variation Point.

pour le prochain top d'horloge ;

– $Cycle_0 : \{s_0\} \times \{v_0\} \rightarrow S \times V$ la fonction qui va définir l'état de la machine à états et sa fonction de valuation en fonction de l'état initial s_0 et v_0. En UML, l'état initial est un pseudo-état muni d'une unique transition sortante sans garde ni événement. Il est donc nécessaire de définir un traitement particulier pour s_0.

Afin de définir formellement le mécanisme des fonctions représentant un cycle de la machine à états, nous établissons également cinq autres fonctions : la fonction d'activité A qui, en fonction de l'état courant, va choisir quelle activité la machine à états doit exécuter ; la fonction de transition T_{tick} portant sur les transitions soumises à l'événement *tick* et la fonction de transition T_c portant sur les transitions soumises à l'événement *completion event* ; les fonctions rtc_{tick} et rtc_c représentant un *Run-to-completion step* correspondant respectivement au traitement de l'événement *tick* et de l'événement *completion event*.

Sémantique

Les fonctions de transition $T_c : S \times V \rightarrow S \cup \{\emptyset\}$ et $T_{tick} : S \times V \rightarrow S \cup \{\emptyset\}$ retournent un nouvel état en accord avec les gardes des transitions. T_c et T_{tick} retournent $\emptyset$ si aucune transition n'est déclenchée. Notons que renvoyer $\emptyset$ ou retourner le même état que celui passé en paramètre n'est pas la même chose. En effet, renvoyer $\emptyset$ signifie qu'aucune transition n'a été prise, retourner le même état signifie qu'une transition réflexive a été prise. Comme un état peut avoir une activité en entrée, une transition réflexive provoquera alors l'exécution de cette activité.

La fonction d'activité $A : S \times V \rightarrow V$ représente l'exécution de l'activité définie en *entry* de chaque état. En fonction de l'état et de la valuation courante des variables du système, la fonction renvoie une nouvelle valuation des variables si une activité est définie pour l'état ou renvoie la même valuation des variables si aucune activité n'est définie. Formellement, si nous notons $S_{activity}$ l'ensemble des états possédant une activitié telle que $S \subseteq S_{activity}$, s un état et v la fonction de valuation des variables du système dans cet état avant l'exécution de A, nous avons :

$$A(s,v) = \begin{cases} v' & \text{si } s \in S_{activity} \text{ , avec } v' \text{une nouvelle valuation des variables} \\ v & \text{sinon} \end{cases}$$

Les fonctions $rtc_c : S \times V \rightarrow (S \cup \{\emptyset\}) \times V$ et $rtc_{tick} : S \times V \rightarrow (S \cup \{\emptyset\}) \times V$ appliquent la fonction de transition, correspondante à l'événement traité, à l'état courant. Si la fonction de transition renvoie $\emptyset$, alors elles retournent $\emptyset$ (la machine à états reste dans le même état), sinon elles appliquent la fonction d'activité A et retournent le nouvel état et la nouvelle fonction de valuation. Plus formellement, si nous notons s un état et v la fonction de valuation des variables du système dans cet état, nous définissons rtc_{tick} comme :

$$rtc_{tick}(s,v) = \begin{cases} (\emptyset, v) & \text{si } T_{tick}(s,v) = \emptyset \\ (T_{tick}(s,v), A(T_{tick}(s,v), v)) & \text{sinon} \end{cases}$$

et similairement rtc_c :

$$rtc_c(s,v) = \begin{cases} (\emptyset, v) & \text{si } T_c(s,v) = \emptyset \\ (T_c(s,v), A(T_c(s,v),v)) & \text{sinon} \end{cases}$$

Le mécanisme de la fonction *Cycle* peut alors être décrit de la manière suivante. Cette fonction commence par traiter les événements *tick* en appelant rtc_{tick}. Si rtc_{tick} renvoie $\emptyset$, cela signifie qu'aucune transition n'a été déclenchée, alors *Cycle* renvoie la fonction identité, que nous notons Id ; si rtc_{tick} retourne un état alors *Cycle* appelle rtc_c tant qu'il y a des transitions déclenchées avec le *completion event*. Formellement, nous avons :

$$Cycle = \begin{cases} rtc_c^n \circ rtc_{tick} & \text{avec } n \in \mathbb{N} \text{ et } rtc_c^{n+1} \text{ renvoyant } \emptyset \\ Id & \text{si } rtc_{tick} \text{ renvoie } \emptyset \end{cases}$$

Notons que nous faisons l'hypothèse qu'il existe, au cours de chaque cycle, un point où plus aucune transition soumise au *completion event* ne peut être déclenchée. Il est à la charge du concepteur de s'assurer de ce point lors de la modélisation.

En se basant sur la sémantique basique du standard UML, le fonctionnement de *Cycle* est justifié de la manière suivante. Lorsqu'un cycle commence, la pile d'événements est vide jusqu'à ce qu'un événement *tick* se produise. Cet événement est alors placé dans la pile. Il est ensuite retiré pour être traité par la machine à états. Si cet événement déclenche une transition, cet événement est consommé, un *Run-to-completion step* est exécuté et son aboutissement va permettre de remplir la pile avec un *completion event*. Ce *completion event* est ensuite retiré et traité par un nouveau *Run-to-completion step*. Si ce traitement permet le déclenchement d'une transition, cet événement est consommé, un *Run-to-completion step* est exécuté et son aboutissement va permettre à nouveau de remplir la pile avec un nouveau *completion event*. La machine à états répète le même mécanisme jusqu'à ce qu'il n'y ait plus de *completion event* dans la pile d'événements (cela correspond à l'appel itératif de la fonction rtc_c dans notre sémantique). En effet, si aucune transition n'est déclenchée suite à un *completion event*, ce *completion event* est abandonné, le *Run-to-completion step* n'est pas exécuté et la pile est donc laissée vide jusqu'au prochain cycle. En se basant sur l'hypothèse de synchronisme posée précédemment, cette chaîne finie de *Run-to-completion step* doit se terminer avant le prochaine cycle i.e. avant qu'un nouvel événement *tick* se produise. En conséquence, au début de chaque cycle, la pile d'événements est toujours vide avant qu'un événement *tick* se produise, et un *completion event* ne se produira qu'après le traitement d'un événement *tick* par un *Run-to-completion step*. La Figure 4.9 décrit un exemple de deux cycles consécutifs dans une machine à états. Notons qu'il s'agit là d'un cas particulier, puisqu'il y a au moins un *Run-to-completion step* pour chaque cycle.

Enfin, nous définissons le comportement du cycle initial d'une machine à états pour un cycle comme : $Cycle_0 = rtc_c^n$ avec $n \in \mathbb{N}^*$ quand rtc_c^{n+1} renvoie $\emptyset$. Cette fonction n'est appelée qu'une seule fois lors du lancement de la machine à états.

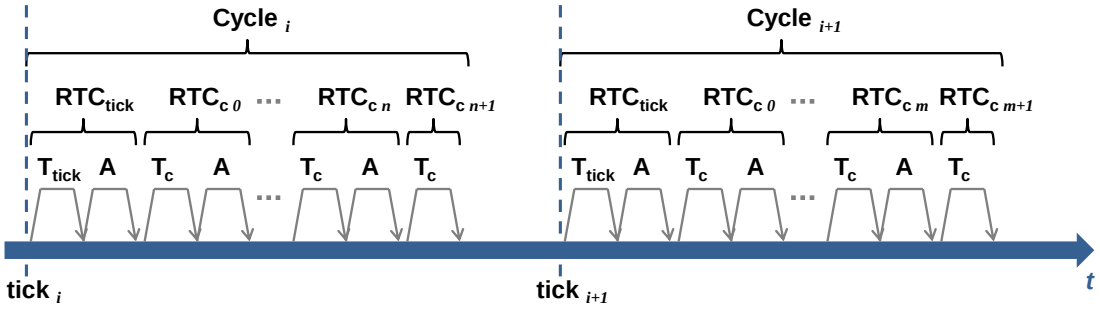


FIGURE 4.9 – Exemple de deux cycles consécutifs

4.3.3 Exemple de modélisation

Afin d'illustrer l'application de nos travaux dans la suite de cette thèse, nous considérons l'exemple donné Figure 4.10 exprimé dans notre sous-ensemble. Il est basé sur un exemple décrit dans [66] qui représente le comportement du logiciel contrôlant le train d'atterrissage d'un drone. Le train d'atterrissage d'un drone est décomposé en trois élé-

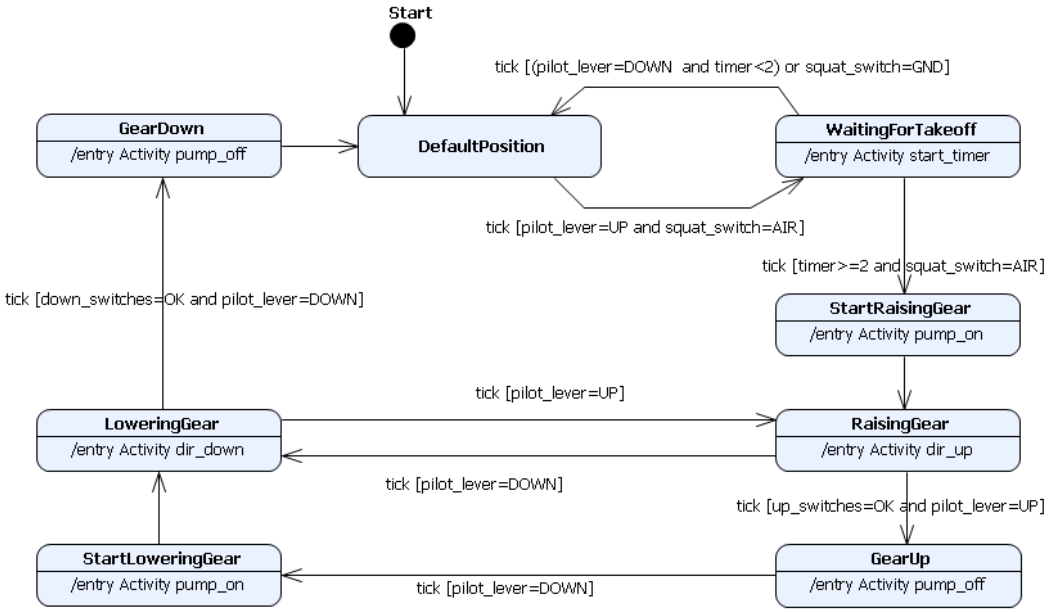


FIGURE 4.10 – Modélisation du train d'atterrissage d'un drone

ments : le train avant, le train droit et le train gauche. Chacun de ces trains est équipé d'un interrupteur de montée et d'un interrupteur de descente, nommé `up_switches` et `down_switches` dans notre exemple. Chaque interrupteur est en position fermé quand le train d'atterrissage est respectivement monté ou descendu. Un interrupteur additionnel

sur le drone, nommé `squat_switch`, indique si le poids du drone est sur son nez ou non. Si le poids est sur le nez, ce qui correspond à `squat_switch=GRD` sur la Figure 4.10, cela signifie que le drone est toujours en contact avec le sol ; sinon, `squat_switch=AIR`, et cela signifie que le drone est dans les airs. Le dépliage et le repliage du train d'atterrissage est géré par une pompe hydraulique contrôlée électriquement. Cette pompe fournit la pression nécessaire aux actionneurs des différents trains. La pression augmente ou diminue suivant une valve contrôlée informatiquement. Quand le pilote veut replier le train d'atterrissage, il lève le levier. Quand le levier est levé, `pilot_lever=UP`, la pompe est activée et la pression est ajustée, ce qui correspond respectivement aux actions `Activity pump_on` et `Activity dir_up`. Quand le pilote veut déplier le train d'atterrissage, il abaisse le levier, `pilot_lever=DOWN`, la pompe est également activée, `Activity pump_on`, et la pression est ajustée, `Activity dir_down`.

Pour donner un exemple de fonctionnement, prenons l'étape de décollage avec un démarrage dans la position par défaut. Le pilote lève le levier et le drone doit être dans les airs pendant au moins deux secondes pour que le train d'atterrissage commence à se replier. Cela permet de s'assurer que le drone ne retouche pas le sol pendant le repliage du train. Après deux secondes, la pompe est donc activée et le train est replié. Une fois le train replié, la pompe est désactivée et le train est dans la position replié.

Chapitre 5

Génération de propriétés comportementales à partir d'un modèle UML

Maintenant que nous avons défini un sous-ensemble des machines à états UML pour la représentation comportementale d'un logiciel au cours de la phase de conception détaillée, nous allons nous appuyer sur les modèles définis à partir de ce sous-ensemble pour en déduire automatiquement des propriétés sous forme d'annotations qui permettront de prouver le code source en utilisant une méthode de preuve déductive. Cette méthode se base sur le langage d'annotations ACSL et sur l'environnement Frama-C. La description de ce langage est donnée dans la Section 5.1.

Dans une démarche IDM classique, les modèles sont utilisés tout au long du cycle de développement et peuvent permettre une génération automatique de code. Dans notre approche, nous considérons que le code peut avoir été produit à partir de son modèle de conception aussi bien par une machine que par un être humain. Mais pour pouvoir générer des annotations de preuve, nous avons besoin de définir un patron pour la structure de code associée à nos machines à états, le respect de cette structure étant indispensable pour notre vérification. Ces travaux sont décrits dans la Section 5.2.

Les annotations que nous générons représentent les propriétés comportementales que le code source doit vérifier afin de prouver sa conformité avec son modèle de conception. Leur déduction à partir du modèle et leur expression sur le code sont décrites dans la Section 5.3.

En complément, l'application de notre méthode sur un exemple est donnée dans la Section 5.4.

5.1 Le langage d'annotations

ACSL [1] est un langage de spécification pour les programmes C. Il se base sur la logique du premier ordre multi-typée [63] et prend notamment en compte les types du langage C tels que les entiers (`integer`) et les réels (`real`) ainsi que les booléens (`boolean`). Les spécification exprimées en ACSL sont représentées par des annotations sur le code ; elles y apparaissent sous forme de commentaires. Afin de les distinguer des commentaires classiques, les annotations ACSL débutent toujours par la balise `@`. Il faut également noter que ces annotations sont sans effet de bord sur le programme.

Différents types d'annotations peuvent être exprimés à partir du langage ACSL :

- Des contrats de fonction : ils permettent de spécifier le comportement attendu d'une fonction. Un contrat de fonction est toujours exprimé avant la fonction sur laquelle il porte.
- Des invariants globaux : ils permettent d'exprimer des propriétés globales sur le programme. Ce type d'annotation est défini au niveau des déclarations dans le programme.
- Des spécifications logiques : ils permettent de définir des lemmes, des axiomes, des prédicats, des fonctions logiques etc. Ces spécifications logiques peuvent ensuite être utilisées dans d'autres annotations. Ces annotations sont généralement exprimées en tête du programme, au niveau des déclarations.
- Des assertions [2] : les assertions permettent de définir des propriétés qui doivent être vraies à certains points du programme. Dans ce sens, ce type d'annotations peut être exprimé n'importe où dans l'algorithme d'une fonction.
- Des annotations de boucle : ces annotations permettent de spécifier le comportement des boucles, notamment à partir de variants et d'invariants. Elles sont principalement utilisées dans le cadre de preuves déductives et s'expriment juste avant la boucle ciblée.
- Du *"ghost code"* : en ACSL, un *"ghost code"* est un bloc d'instructions C exprimé sous forme d'annotations et qui va porter uniquement sur des *"ghost variables"*, c'est-à-dire des variables utilisées uniquement dans le cadre de la spécification d'autres annotations ACSL. Le *"ghost code"* n'influe jamais sur le programme original.

Dans nos travaux, nous sommes principalement intéressés par l'expression de contrats de fonction sur le code. Un contrat de fonction est composé de préconditions et de postconditions portant sur une fonction, similairement au triplet de Hoare donné dans la Section 2.3.2. Sa sémantique est donc la suivante : si les préconditions sont vérifiées à l'appel de la fonction, alors les postconditions doivent être vérifiées après la terminaison de la fonction.

En ACSL, ces préconditions et ces postconditions sont exprimées à l'aide de clauses spécifiques dans les annotations. Une précondition est représentée par une clause *requires*

1. ANSI/ISO C Specification Language.

2. Il s'agit d'assertions au sens défini par le manuel ACSL et qui diffère quelque peu du sens donné par Hoare dans sa logique. Nous avons dit dans la Section 2.3.2 qui décrivait cette logique, que Hoare considérait que les préconditions et les postconditions étaient des assertions.

et une postcondition est représentée par une clause *ensures*. ACSL ajoute également une clause particulière à la représentation classique d'un contrat de fonction : la clause *assigns* permet de garantir dans un contrat que chaque fois que la fonction se termine, seules les zones mémoires spécifiées ont été éventuellement modifiées par cette fonction.

En complément de ces clauses, le langage ACSL définit des mot-clés pour exprimer certains concepts utiles dans l'expression de contrats de fonction. Parmi les plus utiles, le mot-clé $\backslash result$ représente la valeur de retour de la fonction, le mot-clé $\backslash old()$ représente la valeur d'une variable à l'entrée de la fonction et les mots-clé $\backslash true$ et $\backslash false$ représentent les valeurs classiques d'une variable booléenne. Les mots-clé $\backslash result$ et $\backslash old()$ ne peuvent être utilisés que dans des clauses *ensures*.

Un exemple de contrat de fonction est donné Listing 5.1. Le contrat porte sur une fonction prenant en entrée un entier `x` et retournant sa valeur soustraite d'une unité. La précondition du contrat spécifie que l'entier d'entrée de la fonction doit être positif et sa postcondition spécifie que la fonction doit renvoyer le résultat de la soustraction de cet entier.

```
/*@requires x>0;
   ensures \result==\old{x}-1;
*/
int subtract(int x){
    return  x-1;
}
```

Listing 5.1 – Exemple de contrat ACSL

Certaines clauses sont optionnelles dans un contrat de fonction. Par exemple, omettre une clause *requires* dans un contrat revient à spécifier par défaut une clause *requires* avec le mot-clé $\backslash true$. En comparaison, omettre la clause *assigns* dans un contrat de fonction signifie que nous n'avons aucune information sur la modification des allocations mémoire par cette fonction. Cependant, si cette clause *assigns* est spécifiée avec le mot clé $\backslash nothing$, la clause assure qu'aucune allocation mémoire n'a été modifiée.

Outre cette représentation classique d'un contrat de fonction, le langage ACSL permet la définition de contrats plus élaborés. Un contrat de fonction peut être décomposé selon les différents comportements possibles de la fonction analysée, chaque comportement étant alors représenté par un ACSL *behavior*. Cet ACSL *behavior* est composé comme un contrat de fonction classique à partir des clauses définies précédemment, à la différence qu'il s'appuie sur un type de clause supplémentaire. Il s'agit de la clause *assumes* qui va conditionner dans quel cas ce comportement est applicable pour la fonction.

La sémantique d'un *behavior* est donc particulière. Prenons un exemple basique tel que nous pouvons le trouver dans [12]. Cet exemple, décrit Listing 5.2, représente un contrat de fonction composé de deux *behavior* (b1 et b2) chacun composé d'au moins une clause de chaque type. La sémantique de ce contrat est la suivante. L'appelant de la fonction testée doit tout d'abord vérifier que la formule $(A1 ==> R1)\&\&(A2 == R2)$ est vraie à l'appel de la fonction. Ensuite, la fonction testée doit vérifier qu'à la fin de son exécution, la propriété $\backslash old(Ai) ==> Ei$ est vraie pour chaque i. Enfin, la fonction testée doit vérifier pour chaque i, si Ai est vraie à l'appel de la fonction, alors toutes

les allocations mémoire différentes de Vi restent inchangées lors de l'exécution de la fonction.

```
/*@behavior b1 :
      assumes A1;
      requires R1;
      assigns V1;
      ensures  E1;
   behavior b2 :
      assumes A2;
      requires R2;
      assigns V2;
      ensures  E2;
*/
```

Listing 5.2 – Exemple d'ACSL *behavior*

Un exemple sur une fonction concrète est décrit Listing 5.3. Il s'agit d'une fonction permettant de retourner le maximum entre deux entiers `i` et `j`. Il existe deux comportements possibles à cette fonction : soit `i>j`, alors la fonction doit retourner `i` ; soit `j>=i` alors la fonction doit retourner `j`. Dans les deux cas, le contrat spécifie qu'aucune allocation mémoire n'est modifiée (*assigns* *nothing*).

```
/*@behavior i_greaterThan_j :
      assumes i>j;
      assigns \nothing;
      ensures  \result==i;
   behavior j_greaterThan_i :
      assumes j>=i;
      assigns \nothing;
      ensures  \result==j;
*/
int max(int i, int j){
    if (i>j)
        return i;
    else
        return j;
}
```

Listing 5.3 – Exemple d'ACSL *behavior*

ACSL est un langage principalement utilisé dans l'environnement Frama-C. Sa définition est large et certains de ses concepts ne sont pas encore pris en compte par les outils d'analyse statique disponibles dans l'environnement. Cependant, le fragment que nous venons de décrire, limité à un sous-ensemble des contrats de fonction ACSL, est totalement pris en compte par les outils. Nous nous limitons à ce sous-ensemble pour la suite des travaux.

5.2 Un patron pour l'implémentation

5.2.1 La problèmatique

Pour exprimer des annotations ACSL sur un programme, nous avons certes besoin de connaître le comportement attendu de ce programme mais nous avons également besoin

d'informations sur sa structure. Un contrat de fonction est toujours lié à une fonction du programme en particulier et chacune de ses annotations portent sur des variables utilisées par cette fonction. Nous en distinguons deux types : les variables globales et les variables données en paramètre des fonctions analysées. Ces variables peuvent être typées suivant un type du langage C ou suivant un type préalablement défini par l'utilisateur. Notons qu'il est impossible d'écrire un contrat de fonction portant sur des variables locales à la fonction testée [3].

Dans le cadre de nos travaux, le programme à prouver correspond à l'implémentation du comportement d'un logiciel, comportement représenté préalablement par une machine à états. Pour pouvoir écrire des contrats de fonction sur le code source, nous avons alors besoin de connaître :

- la décomposition du programme en terme de fonctions ;
- le prototype de ces fonctions : nom et type des variables données en paramètre et type de retour ;
- le nom et le type des variables globales.

Par ailleurs, plus les fonctions à prouver regrouperont de fonctionnalités, plus les contrats pour ces fonctions seront complexes à exprimer et donc plus difficile sera la preuve. Dans une perspective d'automatisation de la preuve, il est alors crucial de trouver un équilibre entre programme optimisé et programme vérifiable.

5.2.2 Le patron d'implémentation choisi

Pour pouvoir répondre à notre problématique, nous avons défini notre propre patron d'implémentation en nous basant sur la sémantique formelle de notre sous-ensemble. Cette structure permet d'avoir une décomposition modulaire de l'implémentation d'une machine à états et d'être suffisamment proche de la conception détaillée pour favoriser une approche automatique.

Les règles de codage définies sont simples :

- les états de la machine à états sont représentés comme un type énuméré nommé `State`. Les valeurs possibles du type `State` sont tout simplement tous les états possibles de la machine à états auxquels nous rajoutons `Null`. La valeur `Null` nous permet de représenter le $\emptyset$ de notre sémantique ;
- les variables définies dans le modèle conservent le même nom et le même type dans l'implémentation.

L'implémentation est ensuite structurée suivant les différentes fonctions composant la sémantique de nos machines à états :

- Deux fonctions de transition, l'une pour l'événement *tick*, nommée `T_tick`, l'autre pour l'événement *completion event*, nommée `T_c`. Elles permettent de choisir la transition qui sera déclenchée selon la valeur des gardes. Elles renvoient l'état cible si une transition a été déclenchée, `Null` sinon. Ces deux fonctions de transition

3. Les variables définies localement dans une fonction n'ayant pas d'existence avant l'appel de la fonction et après son exécution, elles ne peuvent donc pas être référencées dans des préconditions ou des postconditions.

utilisent une structure classique de la forme `switch/case` pour choisir le comportement associé à l'état courant, puis, pour chaque état, une structure conditionnelle de la forme `if/else` pour choisir quelle transition est déclenchée. Le patron de code pour `T_tick` est donné dans le listing 5.4 (`T_c` est conçue sur le même modèle).

```
State T_tick (State current_state) {
  State output_state = Null;
  switch(current_state) {
    case state1 :
      if (condition_transition1) output_state = targeted_state;
      else if (condition_transition2)
                output_state = other_targeted_state;
    break;
    case state2 :
      if (condition_transition1) output_state = targeted_state;
      else if (condition_transition2)
                output_state = other_targeted_state;
    break;
  }
  return output_state;
}
```

Listing 5.4 – Patron de code pour `T_tick`

– Une fonction d'activité, nommée `A`, qui, pour chaque état, exécute les activités d'entrée de l'état. Conformément à notre sémantique, la fonction `A` ne sera exécutée que si une transition a été déclenchée. Le patron de code pour cette fonction est donné dans le listing 5.5.

```
void A (State current_state) {
  switch(current_state) {
    case state1 :
      Acitivty_state1();
    break;
    case state2 :
      Acitivty_state2();
    break;
    case state3 :
    break;
  }
  return output_state;
}
```

Listing 5.5 – Patron de code pour la fonction d'action `A`

– Deux fonctions de type *run-to-completion* : `RTC_tick` et `RTC_c`, chacune correspondant à un des deux événements possibles. Chacune de ces fonctions appelle la fonction de transition correspondant à son événement. Le patron de code pour la fonction `RTC_tick` est donné dans le listing 5.6 (`RTC_c` est conçue sur le même modèle).

```
State RTC_tick (State current_state) {
  State compute_state = T_tick(current_state);
  if (compute_state != Null) A(compute_state);
  return compute_state;
}
```

Listing 5.6 – Patron de code pour `RTC_tick`

– Une fonction `Cycle` qui implémente le fonctionnement de la machine à états sur un cycle. `Cycle` appelle d'abord la fonction `RTC_tick`, puis, si le retour n'est pas la valeur `Null`, elle appelle la fonction `RTC_c` tant que celle-ci ne renvoie pas `Null`. Rappelons que nous considérons que la terminaison de la fonction doit être vérifiée au niveau du modèle et non au niveau du code. Le concepteur doit s'assurer qu'au cours de chaque cycle, il existe un point où aucune transition sur un *completion event* ne peut être déclenchée. Le patron de code de `Cycle` est donné dans le listing 5.7.

```
State Cycle (State current_state) {
  State compute_state = RTC_tick(current_state);
  State last_state = current_state;
  while (compute_state != Null) {
    last_state = compute_state;
    compute_state = RTC_c(last_state);
  }
  return last_state;
}
```

Listing 5.7 – Patron de code pour `Cycle`

– Une fonction `Cycle_0`, qui correspond au premier cycle de la machine à états (à partir de l'état initial) et qui appelle uniquement la fonction `RTC_c`, conformément à la sémantique de notre sous-ensemble. Le patron de code de `Cycle_0` est donné dans le listing 5.8.

```
State Cycle_0 (State current_state) {
  State compute_state = RTC_c(current_state);
  State last_state = current_state;
  while (compute_state != Null) {
    last_state = compute_state;
    compute_state = RTC_c(last_state);
  }
  return last_state;
}
```

Listing 5.8 – Patron de code pour `Cycle_0`

Le fonctionnement global de la machine à états est finalement implémenté par une boucle infinie dans le programme principal (le `main`). Pour chaque itération de la boucle, le programme attend le prochain *tick* et appelle la fonction `Cycle`. L'attente du prochain *tick* est représentée à partir d'une fonction `wait_tick()`. Le détail de l'implémentation de cette fonction n'est cependant pas nécessaire pour la suite de nos travaux. Le patron de code est donné dans le listing 5.9.

```
current_state = Cycle_0(init_state);
while(1) {
  wait_tick();
  current_state = Cycle(current_state);
}
```

Listing 5.9 – Patron de code du fonctionnement global

5.3 Les annotations générées

La preuve du comportement d'une implémentation consiste à démontrer des propriétés comportementales qui découlent directement de sa conception détaillée réalisée sous forme de machine à états. En pratique, nous allons définir au niveau du code source des contrats sous forme d'annotations ACSL qui correspondent à ces propriétés. Nous pouvons diviser les propriétés comportementales en deux catégories : premièrement celles qui relèvent de la complétude du code vis-à-vis de son modèle de conception : "la conception est totalement implémentée", deuxièmement celles qui relèvent de l'adéquation du code vis-à-vis de son modèle de conception : "seule la conception est implémentée".

Pour une preuve complète de l'implémentation, il faut définir les contrats représentant ces propriétés pour chaque fonction composant l'implémentation. Dans nos machines à états, nous distinguons deux types de fonctions :

- les fonctions représentant le comportement défini par le concepteur. Il s'agit des fonctions de transitions `T_tick` et `T_c` ainsi que de la fonction d'action `A`. Ces fonctions sont propres à chaque machine à états ;
- les fonctions représentant le fonctionnement commun de la machine à états. Il s'agit des fonctions `RTC_tick`, `RTC_c`, `Cycle` et `Cycle_0`. Ces fonctions sont communes à chaque machine à états et le concepteur n'a aucune influence sur leur implémentation.

Nos travaux portent sur la vérification des fonctions représentant la valeur ajoutée par le concepteur. Nous définissons donc pour ces fonctions les contrats qu'il est possible de générer à partir du modèle de conception. La preuve de ces contrats permet alors une preuve partielle de l'implémentation. Nous nous concentrons pour la suite des travaux sur la génération des contrats pour les fonctions de transition. La génération de contrats pour la fonction d'activité a également fait l'objet de travaux mais qui relèvent plus d'une solution d'ingénierie que d'une approche théorique. Une présentation de ces travaux est donnée en Annexe B.

5.3.1 La complétude des fonctions de transition

Pour garantir la complétude par rapport au modèle de conception, les fonctions de transition doivent assurer les propriétés suivantes :

(a) pour l'état courant, si une transition sortante peut-être déclenchée alors la fonction de transition renvoie l'état cible de cette transition ;

(b) la fonction de transition est sans effet de bord.

La propriété (a) est définie par un ACSL *behavior*, un pour chaque état pris en compte par la fonction de transition ciblée [4]. La clause *assumes* du ACSL *behavior* défini porte sur la valeur de l'état courant. Le *behavior* est ensuite composé d'une clause *ensures* pour chaque transition sortante de l'état, transition conditionnée par l'événement correspondant à la fonction de transition ciblée. Le patron de génération de cette propriété est donné Listing 5.10.

4. Un état est dit pris en compte par une fonction de transition si au moins une de ses transitions sortantes est conditionnée par l'événement associé à la transition (tick ou completion event).

```
behavior <nom Etat>
assumes current_state==<nom Etat>;
ensures <Garde de la transition sortante 1>
            ==> \result == <Etat cible de la transition sortante 1>;
   ⋮
ensures <Garde de la transition sortante N>
            ==> \result == <Etat cible de la transition sortante N>;
```

Listing 5.10 – Patron pour la propriété (a)

Nous garantissons la propriété (b) à l'aide d'une clause ACSL de type *assigns* associée au mot clé *nothing* afin de s'assurer qu'aucune allocation mémoire n'est modifiée par la fonction de transition, vérifiant ainsi que la fonction est sans effet de bord. Cette clause est associée à chaque *behavior* défini pour la fonction de transition à prouver. Le patron ACSL de la propriété est donné dans le listing 5.11.

```
assigns \nothing;
```

Listing 5.11 – Patron pour la propriété (b)

5.3.2 L'adéquation des fonctions de transition

L'adéquation d'une fonction de transition vis-à-vis du modèle de conception signifie ici que seules les transitions spécifiées, et aucune autre, sont implémentées dans le programme. Pour garantir cela, l'implémentation doit assurer les propriétés suivantes :

(c) pour l'état courant, si une transition a été déclenchée alors sa garde doit-être vraie ;

(d) pour l'état courant, si aucune garde d'aucune transition sortante n'est vraie, alors aucune transition n'est déclenchée i.e. la fonction de transition renvoie $\emptyset$;

(e) si un état n'est pas pris en compte par une fonction de transition (aucune transition sortante de cet état n'est déclenchée par l'événement géré par cette fonction), cette fonction de transition ne déclenche aucune transition pour cet état i.e. la fonction de transition renvoie $\emptyset$.

La propriété (c) est définie par un ACSL *behavior*, un pour chaque état pris en compte par la fonction de transition ciblée. La clause *assumes* du ACSL *behavior* défini porte sur la valeur de l'état courant. Le *behavior* est ensuite composé d'une clause *ensures* pour chaque transition sortante de l'état, transition conditionnée par l'événement correspondant à la fonction de transition ciblée. Le patron de génération de cette propriété est donné en Listing 5.12.

```
behavior <nom Etat>
assumes current_state==<nom Etat>;
ensures \result == <Etat cible de la transition sortante 1>
            ==> <Garde la transition sortante 1>;
   ⋮
ensures \result == <Etat cible de la transition sortante N>
            ==> <Garde la transition sortante N>;
```

Listing 5.12 – Patron pour la propriété (c)

La propriété (d) est également définie par un ACSL *behavior*, un pour chaque état pris en compte par la fonction de transition ciblée. La clause *assumes* de ce *behavior* porte sur la valeur de l'état courant. Le *behavior* est ensuite composé d'une unique clause *ensures* représentant que si aucune garde des transitions sortantes de l'état n'est vraie, la fonction de transition doit alors retourner la valeur `Null` i.e. aucune transition n'est déclenchée. Le patron de génération est donné dans le Listing 5.13(en ACSL, le symbole " !" correspond à la négation logique).

```
behavior <nom Etat>
assumes current_state==<nom Etat>;
ensures (!<Garde la transition sortante 1>
          && ...
          && !<Garde la transition sortante N>)
                    ==> \result == Null;
```

Listing 5.13 – Patron pour la propriété (d)

La propriété (e) signifie que la valeur de retour de la fonction de transition doit être `Null` pour tous les états qui ne sont pas gérés. Elle est générée en ACSL sous la forme d'un unique *behavior* pour chaque fonction de transition. Ce *behavior* est composé d'une clause *assumes* portant sur la valeur de l'état courant et d'une clause *ensures* garantissant que la valeur `Null` est renvoyée par la fonction de transition dans le cas où l'état courant ne correspond à aucun état défini dans le modèle. Le patron de génération est donné dans le Listing 5.14.

```
behavior OtherStates :
    assumes current_state != <Etat 1>
              && ...
                 && current_state != < Etat n>;
    assigns \nothing;
    ensures \result == Null;
```

Listing 5.14 – Patron pour la propriété (e)

En pratique, les patrons de générations des propriétés (a), (b), (c) et (d) peuvent être fusionnés dans un seul *behavior* pour chaque état pris en compte de la fonction de transition analysée. En effet, les noms des *behavior* et leur clause *assumes* sont identiques pour les propriétés (a), (c) et (d). Seules les clauses *ensures* varient et peuvent alors être regroupées au sein d'un même *behavior*. Le patron de génération global pour ces propriétés est donné Listing 5.15. La propriété (b) est présente grâce à la clause *assigns*. Les clauses *ensures* issues des patrons des propriétés (a) et (c) ont pu être fusionnées sous la forme d'une clause *ensures* pour chaque transition sortante et représentant une équivalence portant sur la condition de cette transition et de son état cible. Enfin la propriété (d) est représentée par la dernière clause *ensures* du *behavior*.

```
behavior <nom Etat> :
    assumes current_state == nom Etat;
    assigns \nothing;
    ensures <Garde de la transition sortante 1>
            <==> \result == <Etat cible de la transition sortante 1>;
    .
    .
    .
    ensures <Garde de la transition sortante N>
            <==> \result == <Etat cible de la transition sortante N>;
```

```
    ensures (!<Garde la transition sortante 1>
          && ...
          && !<Garde la transition sortante N>)
                    ==> \result == Null;
```

Listing 5.15 – Patron global pour les propriétés (a) (b) (c) et (d) pour un état

La définition de ces *behavior* pour chaque état pris en compte par la fonction de transition ainsi que la définition du *behavior* représentant la propriété (e) composent le contrat de fonction à vérifier sur cette fonction de transition. Notons que même si avec notre méthode nous sommes en mesure de détecter des transitions non spécifiées, nous ne pouvons pas détecter le code mort, c'est-à-dire du code qui ne s'exécute jamais et qui, ici, implémenterait des transitions qui ne sont jamais déclenchées ou des états jamais atteints.

5.4 Application sur un exemple

Pour illustrer notre approche, nous appliquons notre patron d'implémentation et nos patrons de propriétés pour les fonctions de transition de l'exemple du train d'atterrissage (Figure 4.10 donnée dans la Section 4.3.3).

5.4.1 Implémentation des fonctions de transitions

Le code C correspondant à l'implémentation de la fonction `T_tick` de l'exemple est donnée Listing 5.16.

```
State T_tick(State current_state){
  State output_state=Null;
  switch(current_state) {
    case DefaultPosition:
            if (pilot_lever==UP && squat_switch==AIR)
                output_state=WaitingForTakeoff;
    break;
    case WaitingForTakeoff:
            if (timer>=2 && squat_switch==AIR) output_state=StartRaisingGear;
            else if ((pilot_lever==DOWN && timer<2)||squat_switch==GND)
                      output_state=DefaultPosition;
    break;
    case RaisingGear:
            if (pilot_lever==DOWN) output_state=LoweringGear;
            else if (pilot_lever==UP && up_switches==OK) output_state=GearUp;
    break;
    case GearUp:
            if (pilot_lever==DOWN) output_state=StartLoweringGear;
    break;
    case LoweringGear:
            if (pilot_lever==UP) output_state=RaisingGear;
            else if (pilot_lever==DOWN && down_switches==OK)
                      output_state=GearDown;
    break;
  }
  return output_state;
}
```

Listing 5.16 – Implémentation de la fonction T_{tick} pour l'exemple de la Figure 4.10

5.4.2 Les annotations générées

Le contrat représentant les propriétés de complétude et d'adéquation issues du modèle pour la fonction `T_tick` est disponible Listing 5.17. Ce contrat se base sur les patrons de propriétés définies précédemment. Prenons par exemple le premier *behavior* défini dans le contrat. Il concerne l'état `DefaultPosition` de l'exemple. La clause '`assigns \nothing;`" represente la propriété (b). Ensuite, puisqu'il n'y a qu'une seule transition sortante de l'état `DefaultPosition` déclenchable par l'événement *tick*, les propriétés (a) et (b) sont définies par la clause "`pilot_lever == UP && squat_switch == AIR) <==> \result == WaitingForTakeoff;`". La clause "`! (pilot_lever == UP && squat_switch == AIR) ==> \result == Null;`" represente la propriété (d). Enfin, la propriété (e) est représentée par le *behavior* nommé `OtherStates`.

```
/*@behavior DefaultPosition :
    assumes current_state == DefaultPosition;
    assigns \nothing;
    ensures (pilot_lever == UP && squat_switch == AIR)
                <==> \result == WaitingForTakeoff;
    ensures  ! (pilot_lever == UP && squat_switch == AIR) ==> \result == Null;

behavior WaitingForTakeoff :
    assumes current_state == WaitingForTakeoff;
    assigns \nothing;
    ensures ( ( pilot_lever == DOWN && timer < 2 ) || squat_switch == GND)
                <==> \result == DefaultPosition;
    ensures (timer >= 2 && squat_switch == AIR)
                <==> \result == StartRaisingGear;
    ensures  ! ( ( pilot_lever == DOWN && timer < 2 )  || squat_switch == GND)
                &&  ! (timer >= 2 && squat_switch == AIR)
                    ==> \result == Null;

behavior RaisingGear :
    assumes current_state == RaisingGear;
    assigns \nothing;
    ensures (pilot_lever == DOWN) <==> \result == LoweringGear;
    ensures (up_switches == OK && pilot_lever == UP) <==> \result == GearUp;
    ensures  ! (pilot_lever == DOWN)
              &&  ! (up_switches == OK
                    && pilot_lever == UP) ==> \result == Null;

behavior GearUp :
    assumes current_state == GearUp;
    assigns \nothing;
    ensures (pilot_lever == DOWN) <==> \result == StartLoweringGear;
    ensures  ! (pilot_lever == DOWN) ==> \result == Null;

behavior LoweringGear :
    assumes current_state == LoweringGear;
    assigns \nothing;
    ensures (pilot_lever == UP) <==> \result == RaisingGear;
    ensures (down_switches == OK && pilot_lever == DOWN)
              <==> \result == GearDown;
    ensures  ! (pilot_lever == UP)
              &&  ! (down_switches == OK
                    && pilot_lever == DOWN) ==> \result == Null;

behavior OtherStates :
    assumes current_state != DefaultPosition
```

```
              && current_state != WaitingForTakeoff
                  && current_state != RaisingGear
                      && current_state != GearUp
                          && current_state != LoweringGear;
    assigns \nothing;
    ensures \result == Null;
*/
State T_tick(State current_state){
            ⋮
```

Listing 5.17 – Annotations générées pour la fonction T_{tick}

5.5 Synthèse

La contribution que nous avons proposée ([54, 55]) permet de prouver automatiquement la conformité de l'implémentation des fonctions de transition par rapport à un modèle de machine à états UML. Elle garantit la complétude et l'adéquation de l'implémentation des fonctions de transition d'une machine à états par rapport à son modèle UML. Elle va donc permettre de détecter des erreurs d'implémentation du modèle, mais également des comportements ajoutés dans l'implémentation et non spécifiés dans le modèle. Pour les fonctions de transition, ces ajouts peuvent être l'implémentation d'une transition non spécifiée ou l'implémentation d'un état non spécifié qui vont modifier le comportement attendu du programme.

Notre méthode présente également des limites :

- il n'est pas possible de détecter du code mort. Le code mort est, par définition, une portion de programme qui ne sera jamais exécutée. Il n'influe pas sur le comportement attendu du programme et ne peut donc pas enfreindre les contrats de fonction à vérifier que nous définissons ;
- le modèle de conception doit être correct afin de pouvoir vérifier un code correct. Si par exemple le modèle définissait une séquence infinie de transitions déclenchées uniquement par des *completion event*, le code qui découlerait de ce modèle ne pourrait jamais se terminer [5]. Il n'y aurait certes pas d'impact sur la vérification des fonctions de transition, mais la vérification des autres fonctions implémentant la machine à états serait compromise par leur non-terminaison ;
- dans le modèle de conception, l'utilisateur ne peut pas spécifier deux transitions sortantes d'un même état, spécifiées avec deux gardes disjointes mais déclenchées par le même événement et ayant pour cible le même état. Si c'était le cas, nos contrats ne pourraient pas être vérifiés. Prenons par exemple la machine à états très simpliste définie Figure 5.1.

  Cette machine à états est composée de deux états, *S1* et *S2*, et *S1* possède deux transitions sortantes, déclenchées par l'événement *tick*, dont les gardes sont mutuellement disjointes et qui ont toutes les deux pour cible *S2*. Ce modèle est sé-

5. La boucle `while` de la fonction `Cycle` ne se terminerait jamais et la machine à états serait infiniment bloquée dans un seul cycle.

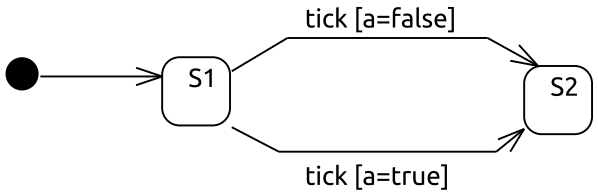


FIGURE 5.1 – Exemple de machine à états non prise en charge

mantiquement correct : les deux transitions indiquent simplement que quelle que soit la valeur de *a*, *S1* ira en *S2* au prochain *tick*.
Le code défini Listing 5.18 est également juste : quelle que soit la valeur de `a`, la fonction de transition renverra `S2` à son prochain appel si l'état courant est `S1`. Cependant, notre contrat de fonction, défini également Listing 5.18, ne peut pas être vérifié : notre contrat spécifie que nous devons vérifier que ($a == true$ <==> $output\_state = S2$) et ($a == false$ <==> $output\_state = S2$) ce qui n'est pas possible. Il faut donc être capable de factoriser les transitions avant de permettre la génération des contrats.

```
/*@behavior S1 :
assumes current_state==S1;
assigns \nothing;
ensures a==true <==>  output_state=S2;
ensures a==false <==>  output_state=S2;
ensures !(a==true) &&  !(a==false) ==>  output_state=Null;
*/
State T_tick(State current_state){
  State output_state=Null;
  switch(current_state) {
    case S1 :
            if a==true output_state=S2;
            else if  a==false output_state=S2;
    break;
  }
  return output_state;
}
```

Listing 5.18 – Contrats et implémentation de la fonction de transition `T_tick` de la machine à états définie Figure 5.1

Chapitre 6

Prototypage et évaluation de l'approche

Nous avons défini des patrons de génération d'annotations ACSL pour les propriétés que nous souhaitons vérifier. Ces patrons présentent l'avantage de pouvoir être utilisés dans une approche totalement automatique. En s'appuyant sur le type d'outillage utilisé au sein des équipes Atos, nous avons implémenté notre approche dans un prototype nommé AGrUM [1] présenté dans la Section 6.1.

Suite à cette implémentation, nous avons évalué et comparé notre approche par rapport à plusieurs points. Nous avons tout d'abord comparé dans la Section 6.2, notre contribution avec l'approche attendue au début de cette thèse. Nous avons ensuite étendu nos refléxions aux attentes d'une partie de la communauté informatique française en proposant une enquête sur l'usage des méthodes formelles pour la vérification de programmes. Les résultats de cette enquête et le parallèle avec notre approche sont décrits dans la Section 6.3. Enfin, la Section 6.4 propose un résumé des travaux existants et leur comparaison avec ceux réalisés dans cette thèse.

6.1 Le prototype AGrUM

AGrUM [53] est un greffon Eclipse [2] pour la génération automatique d'annotations de preuve en ACSL à partir d'un modèle de machine à états UML vers un programme C. Eclipse est une plate-forme libre et open-source développée en Java. Elle a été mise au point afin de fournir un environnement de développement extensible pour l'ingénierie logicielle et regroupe aujourd'hui un ensemble de greffons offrant différentes activités telles que le développement en Java, le développement en C, la gestion de versions, la création de clients riches, etc. Le choix de cet environnement pour la définition de notre outil s'est imposé face aux activités menées par Atos dans le cadre du développement de systèmes et de logiciels embarqués. Leur conception y est notamment réalisée à partir de l'envi-

1. ACSL Generator from UML Model.
2. `www.eclipse.org`

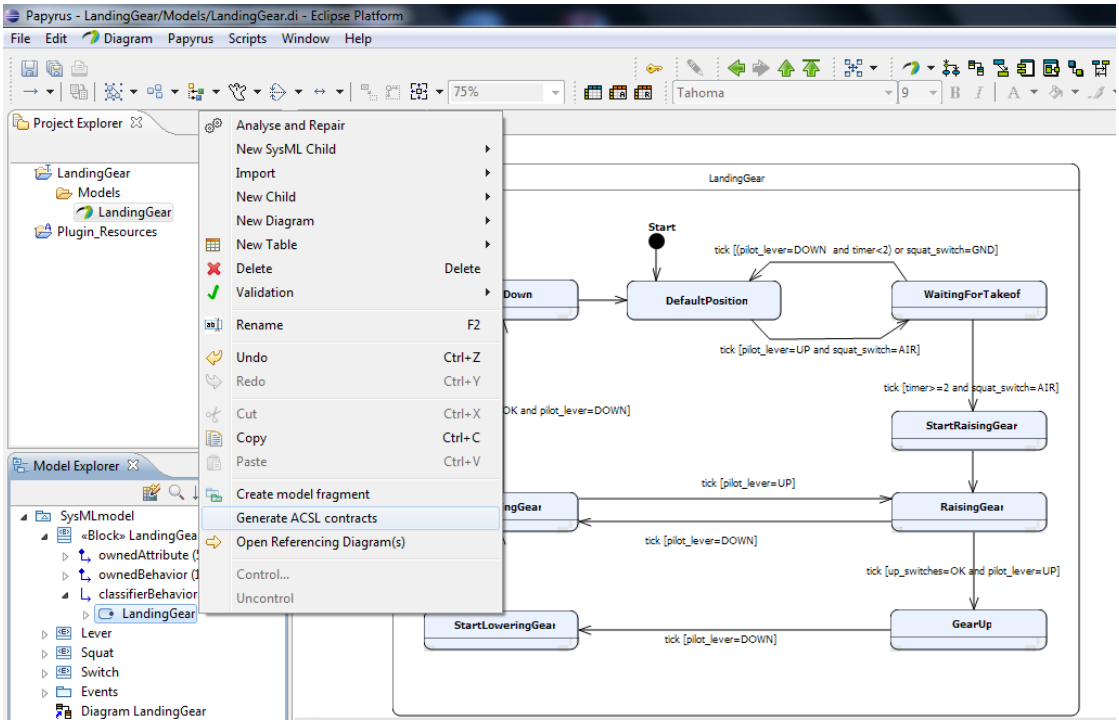


FIGURE 6.1 – Commande du greffon AGrUM dans le menu de l'environnement Papyrus

ronnement TOPCASED. Cet environnement est une plate-forme client riche Eclipse et maintenu lors de ma thèse par les équipes Atos. Dans cet environnement, la modélisation UML/SysML des systèmes et des logiciels embarqués est réalisable à partir des éditeurs fournis par le greffon Papyrus MDT [3] pour l'édition de modèles UML/SysML.

L'outil AGrUM s'appuie donc sur ces technologies en prévision d'une intégration naturelle dans les environnements de développement connus et maîtrisés au sein d'Atos et par une partie de leurs clients. Il peut être installé sur n'importe quel type de plate-forme Eclipse version 3.7.x ainsi que sur la plate-forme TOPCASED version 5.2.1 ou plus récente. Son fonctionnement nécessite l'installation du greffon Papyrus MDT, qui est automatique lors de l'installation d'AGrUM.

L'utilisation d'AGrUM est simple. Au sein de l'interface de travail fournie par le greffon Papyrus MDT, l'utilisateur a accès à son projet de modélisation. Ce projet peut contenir un ou plusieurs modèles. Afin d'utiliser le greffon AGrUM, l'utilisateur doit simplement sélectionner la machine à états pour laquelle il souhaite générer des annotations de preuve à partir de l'explorateur de modèles [4] fourni par le greffon Papyrus MDT et une commande [5] devient alors disponible pour la génération d'annotations ACSL. Un aperçu est disponible Figure 6.1.

3. `www.eclipse.org/papyrus/`
4. Vue "Model explorer".
5. "Generate ACSL contracts".

La génération des annotations est réalisée en 4 étapes :

1. AGrUM vérifie que la machine à états qui a été sélectionnée est conforme à notre sous-ensemble UML. Actuellement, AGrUM vérifie qu'il n'y a qu'une seule région définie pour la machine à états, qu'il existe un unique pseudo-état et qu'il est de type Initial. Il vérifie également qu'il n'y a qu'une seule transition sortante de cet état et qu'elle ne contient aucun *Trigger* et aucune *Guard* ;
2. le greffon vérifie ensuite que le fichier de code source C donné contient bien les fonctions pour lesquelles il doit générer des annotations. En se basant sur le patron d'implémentation défini en Section 5.2, cette vérification consiste à vérifier la signature des fonctions du programme. Actuellement, AGrUM ne vérifie que les signatures des fonctions de transition. Le parcours du fichier C est réalisé grâce la librairie Eclipse CDT [6] qui permet d'obtenir l'arbre de syntaxe abstraite du fichier. Les signatures peuvent ensuite être recherchées dans cet arbre et leurs positions dans le fichier sont mémorisées afin qu'AGrUM connaissent les emplacements où les annotations doivent être générées ;
3. la machine à états est analysée afin de créer les contrats de fonction à partir des patrons de génération précédemment définis. Cette analyse est réalisée à partir des librairies disponibles dans le greffon Papyrus pour la représentation du modèle en Java et à partir de la librairie Guava [7] qui offre un ensemble d'utilitaires pour l'écriture de programme Java. Cette librairie est principalement utilisée dans AGrUM pour faciliter la création d'itérateurs répondant à certains prédicats afin de parcourir la collection d'éléments constituant le modèle. La création des contrats de fonction est réalisée suite à l'analyse de la machine à états. Notons que pour l'analyse des gardes exprimées en OCL dans la machine à états, nous avons créé un parser spécifique pour le traitement du sous-ensemble que nous utilisons. Ce parser a été créé en utilisant la technologie ANTLR [8] qui permet la génération automatique de parsers à partir d'une grammaire ;
4. une fois les contrats créés, AGrUM génère un nouveau fichier C, correspondant au code C précédemment analysé auquel est rajouté les annotations générées aux emplacements requis.

La génération des annotations est implémentée en Java. Une autre approche aurait été d'implémenter notre approche par des transformations de modèles. Nous aurions pu transformer un modèle de machine à états UML vers un modèle de contrats de fonction ACSL ou transformer un modèle de machine à états UML vers du texte en utilisant par exemple la technologie Eclipse Model-To-Text [9] citée Section 2.1.3. Notons que bien que le méta-modèle UML soit défini dans Eclipse, il n'existe pas de méta-modèle pour le langage ACSL. Le choix de passer par une génération des annotations ACSL purement implémentée en Java a été fait : premièrement, dans un souci d'obtenir un démonstrateur

6. C/C++ Development Tooling. `www.eclipse.org/cdt/`
7. `https://code.google.com/p/guava-libraries`
8. `http://www.antlr.org/`
9. `http://www.eclipse.org/modeling/m2t/`

pratique de l'approche à l'intention de futurs utilisateurs dans un impératif de temps court ; deuxièmement afin de rendre l'implémentation abordable et maintenable par des développeurs Java sans connaissance des techniques de transformation de modèles.

Le prototype AGrUM est libre et open-source. Il est disponible à l'adresse `http://code.google.com/a/eclipselabs.org/p/agrum/`. Ce site contient de la documentation, un exemple, le code source, une vidéo de démonstration ainsi qu'une archive d'installation.

6.2 Evaluation par rapport à l'objectif de départ

6.2.1 Une approche outillée pour des utilisateurs non experts

L'approche théorique proposée a l'avantage d'être automatisable au sein de l'outil AGrUM compatible avec l'environnement de travail d'une partie des utilisateurs soutenus par Atos. En reprenant la Figure 3.3 décrite dans le Chapitre 3, nous définissons alors la chaine d'outil optimale couvrant notre approche dans la Figure 6.2.

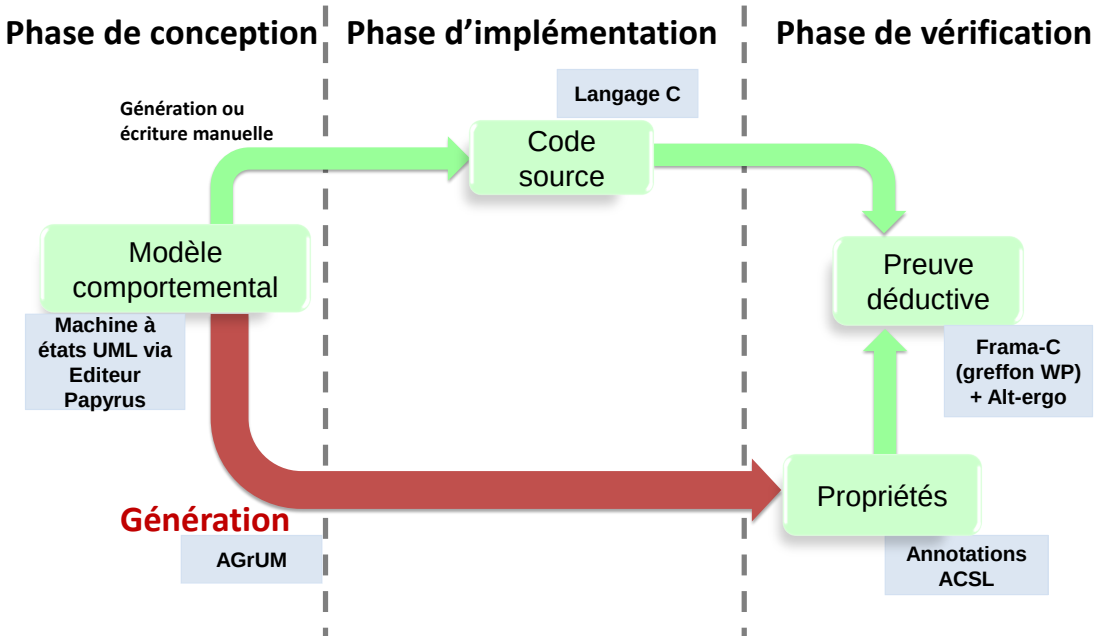


FIGURE 6.2 – Chaîne de vérification outillée

Dans la phase de conception, le modèle comportemental est réalisable à partir des éditeurs Papyrus sous la forme d'une machine à états UML. Durant la phase d'implémentation, le code source est obtenu à partir de ce modèle dans le langage C. Il peut être écrit manuellement ou généré par un générateur arbitraire, tant que le patron d'implémentation est respecté. La phase de vérification se base sur la preuve déductive du code source à partir des contrats de fonction générés sous forme d'annotations ACSL dans le modèle. La génération des annotations est bien entendu gérée par l'outil AGrUM que

nous venons de décrire. La preuve déductive est automatiquement réalisée par l'environnement Frama-C et son greffon WP.

Ce greffon permet de calculer des obligations de preuve sur un programme C annoté par calcul de la plus faible précondition, technique introduite par [39]. Ces obligations de preuve peuvent ensuite être envoyées à un solveur qui va permettre de statuer sur la validité des annotations dont ces obligations sont dérivées. Le processus de vérification lié à l'utilisation de ce greffon est décrit Figure 6.3.

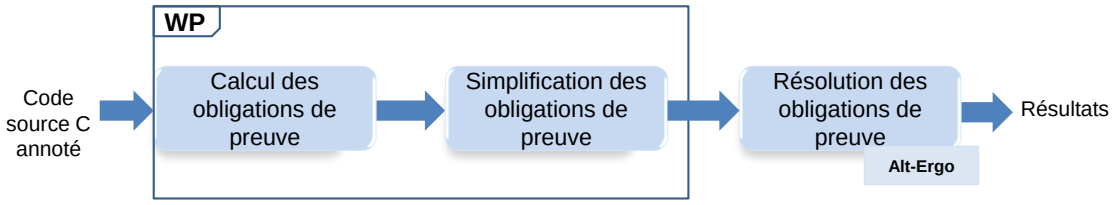


FIGURE 6.3 – Processus de vérification de Frama-C à partir du greffon WP

Le calcul des obligations de preuve réalisé par WP peut être décomposé en deux étapes :

- La première étape représente le calcul basique des obligations de preuve. Elle consiste à transformer le code C annoté, avec des spécifications ACSL, vers des formules logiques basées sur la logique du premier ordre. Cette transformation se base sur le modèle mémoire choisi pour effectuer le calcul des obligations de preuve. Un modèle mémoire définit la représentation logique des différents types, variables et données utilisés par un programme lors de son exécution. Plus le modèle mémoire est précis, plus les obligations de preuve qui seront générées à partir du programme annoté seront complexes. Le greffon WP propose un modèle mémoire standard par défaut ainsi que d'autres modèles mémoire plus complexe. Pour plus de détails, le lecteur pourra consulter [11].
- Les obligations de preuve générées sont ensuite simplifiées à partir d'un moteur de simplification implémenté dans WP et nommé Qed. Les simplifications sont menées grâce à des régles de normalisation et des lois de simplification implémentées dans le moteur Qed. Pour plus de détails, le lecteur pourra consulter [11].

Les obligations de preuve simplifiées sont ensuite transmises à un solveur afin d'être évaluées. Le greffon WP est nativement configuré pour fonctionner avec le solveur Alt-Ergo [10]. C'est un solveur SMT open-source dédié à la preuve de formules mathématiques établies dans le cadre de la vérification de programmes. Lors de son utilisation avec le greffon WP, les résultats obtenus pour chaque obligation de preuve sont transmis à Frama-C afin de valider ou non les annotations définies sur le programme.

Ainsi, l'utilisateur dispose d'une chaîne d'outil complète qui automatise la preuve partielle d'un code C par rapport à un modèle de machine à états UML, conformément à l'axe de recherche défini Section 3.4.2. Par exemple, l'application de cette chaîne d'outil

10. `alt-ergo.lri.fr`

sur l'exemple décrit Figure 4.10 et Section 5.4 a permis de générer automatiquement la cinquantaine de lignes de contrats de fonction pour la quarantaine de lignes de code C représentant les fonctions de transition et de vérifier l'ensemble de ces contrats en quelques secondes [11].

Nous pouvons toutefois nous poser deux questions majeures concernant notre approche. Nous pouvons nous interroger sur les différences et les apports de notre approche par rapport à une génération automatique de code à partir du modèle de conception. D'un point de vue technique, générer des annotations est assez similaire à générer du code, mais, dans un cadre de certification comme celui de la DO-178 qui entoure nos travaux, les contraintes de qualification pour un outil de vérification sont beaucoup plus simples que celles pour un générateur de code. En effet, si l'outil de vérification fonctionne mal, il n'introduira pas d'erreurs dans le code contrairement à un générateur. Par conséquent, un générateur de code devra être qualifié au moins au même niveau de criticité que le logiciel qu'il génère, ce qui n'est pas le cas pour un outil de vérification.

Nous pouvons également nous demander si cette chaîne outillée permet réellement l'usage d'une vérification formelle par des non experts. Dans le cadre de la preuve du code, si toutes les annotations sont validées, l'utilisateur à la confirmation que ce qu'il a implémenté correspond à la conception. Cependant, si des annotations ne sont pas validées, il est aujourd'hui à la charge de l'utilisateur de trouver l'erreur dans l'implémentation en analysant les annotations non valides, annotations exprimées en ACSL.

6.2.2 Réponse à la DO-178 et à la DO-333

Les objectifs de vérification à atteindre pour la certification sont définis sur la Figure 3.1. Dans la norme de certification, ces objectifs sont récapitulés pour chaque phase de développement dans des tables données en annexe de la norme. Ces tables référencent les sections du document où les objectifs sont définis. Il existe dix tables au total.

Notre approche concerne la vérification de la conformité d'un code source par rapport à sa conception. Dans le cadre de la DO-333 et de l'usage des méthodes formelles, cet objectif s'inscrit dans l'objectif de vérification concernant la conformité du code exécutable par rapport à la conception, comme décrit Figure 3.2. Notre approche est donc liée à quatre tables :

- *Table FM.A-4 : vérification des sorties de la phase de conception logicielle* [12] ;
- *Table FM.A-5 : vérification des sorties des phases d'implémentation et d'intégration logicielle* [13] ;
- *Table FM.A-6 : test des sorties de la phase d'intégration logicielle* [14] ;
- *Table FM.A-7 : vérification des résultats de la phase de vérification* [15].

11. La vérification a été réalisée avec Frama-C version Fluorine, WP version 0.7 et son modèle mémoire Typed.
12. En anglais : Verification of Outputs of the Software Design Process.
13. En anglais : Verification of Outputs of Software Coding and Integration Processes.
14. En anglais : Testing of Outputs of Integration Process.
15. En anglais : Verification of Verification Process results.

Concernant les tables restantes, les tables FM.A-1, FM.A-8, FM.A-9 et FM.A-10 traitent de la planification, de la gestion de configuration, de l'assurance qualité et des consultants de certification. La table FM.A-2 définit les objectifs pour les processus de développement et la table FM.A-3 traite de la vérification des résultats du processus de définition des exigences logicielles.

Objectifs atteints

Pour chaque objectif, nous donnons dans [56] la référence à l'objectif à l'intérieur de la table, puis nous listons les sous-objectifs dans les sections liées et les justifications associées à notre approche.

1. *Table FM.A-4, Objectif FM4 : les exigences de bas niveaux sont vérifiables*[16]
   - *Verifiabilité (FM.6.3.2.d) :* la conception est exprimée dans un sous-ensemble restreint des machines à états UML qui peut être transformé dans des annotations ACSL et vérifié sur le code. Par conséquent, la conception est vérifiable.
2. *Table FM.A-5, Objectif FM1 : le code source est conforme aux exigences de bas niveau*[17]
   - *Conformité aux exigences de bas niveau (FM.6.3.4.a)* : la conformité du code avec la conception est accomplie par l'utilisation de deux outils. AGrUM qui traduit la conception UML en annotations ACSL et Frama-C qui vérifie que le code source satisfait les annotations ACSL. Ces deux outils doivent être qualifiés avec la DO-330 (Document pour la qualification des outils).
   - *Vérifiabilité (FM.6.3.4.c)* : un patron d'implémentation particulier est défini dans notre approche dans le but d'assurer que le code est compatible avec l'outil de vérification.
   - *Traçabilité (FM.6.3.4.e)* : la vérification du code source par rapport à la conception garantit que la conception se retrouve dans le code source.
3. *Table FM.A-5, objectif FM13 : la méthode formelle est correctement définie, justifiée et appropriée*[18] Cet objectif supplémentaire doit être atteint chaque fois qu'une méthode formelle est utilisée pour répondre à un objectif de vérification. Il est divisé en sous-objectifs :
   - *Notations formelles (FM.6.2.1.a)* : la syntaxe et la sémantique du sous-ensemble du langage de programmation C sont définies mathématiquement. Il est précis et non-ambigu. Nous avons également défini la sémantique formelle du sous-ensemble UML que nous utilisons dans notre approche.
   - *Correction (FM.6.2.1.b)* : la technique de vérification utilisée dans Frama-C est basé sur la logique de Hoare qui a été prouvée correcte à condition que la logique sous-jacente utilisée le soit également[76].
   - *Hypothèses (FM.6.2.1.c)* : aucune hypothèse particulière n'est définie pour la vérification des propriétés sur le code source.

16. En anglais : Low-level requirements are verifiable.
17. En anglais : Source code complies with low-level requirements.
18. En anglais : formal method is correctly defined, justified and appropriate.

4. *Table FM.A-6, Objectif FM3 : Le code exécutable est compatible avec la conception*[19]
   - *Conformité avec la conception (FM.6.7.d)* : L'analyse formelle du code source est réalisée afin d'atteindre cet objectif. Des analyses complémentaires sont cependant nécessaires pour montrer la préservation des propriétés entre le code source et le code executable.
5. *Table FM.A-6, Objectif FM4 : Le code exécutable est robuste par rapport à la conception*[20]
   - *Robustesse avec la conception (FM.6.7.b)* : L'analyse formelle du code source est réalisée afin d'atteindre cet objectif. Des analyses complémentaires sont cependant nécessaires pour montrer la préservation des propriétés entre le code source et le code executable.

Objectifs restants à atteindre

Notre approche permet de répondre à certains objectifs de la DO-178 pour la vérification de la conformité du code source par rapport à la conception. Cependant, pour pouvoir y répondre totalement, notre approche doit encore atteindre d'autres objectifs [56] :

- *Table FM.A-5, Objectif FM10 : les analyses formelles des différents cas et procédures sont correctes*[21] ;
- *Table FM.A-5, Objectif FM11 : les résultats de l'analyse formelle sont corrects et les divergences sont expliquées*[22] ;
- *Table FM.A-7, Objectif FM4 : la couverture des exigences de bas niveau est atteinte*[23] ;
- *Table FM.A-7, Objectif FM5-8 : la vérification de la structure logicielle est atteinte*[24] ;
- *Table FM.A-7, Objectif FM9 : la vérification de la préservation des propriétés entre le code source et le code exécutable est atteinte*[25].

Les trois premiers peuvent être facilement atteints par des revues. Les deux derniers sont les plus difficles à réaliser. Ils nécessitent des analyses spécifiques qui restent à définir. Les objectifs FM5 à FM8 dans la table FM.A-7 sont les objectifs qui remplacent les objectifs de couverture structurelles utilisés lors de l'utilisation de tests. Dans le cadre de l'utilisation des méthodes formelles, l'objectif de couverture est remonté au niveau des phases de spécification et de conception. Nous devons alors démontrer à ces niveaux que la conception qui a été prouvée formellement est complète par rapport au code.

19. En anglais : Executable object code complies with low-level requirements.
20. En anglais : Executable object code is robust with low-level requirements.
21. En anglais : Formal analysis cases and procedures are correct.
22. En anglais : Formal analysis results are correct and discrepancies explained.
23. En anglais : Coverage of low-level requirements is achieved.
24. En anglais : Verification of software structure is achieved.
25. En anglais : Verification of property preservation between source and object code.

Pour aller plus loin et obtenir une vérification de programme complète, l'objectif FM9 de la table FM.A-7 reste à atteindre puisque les méthodes formelles sont appliquées sur le code source et non sur le code objet exécutable. Il faut donc montrer que les propriétés vérifiées sur le code sont toujours vérifiées sur le code exécutable. Cet objectif peut être atteint par des revues ou des techniques formelles [85].

6.3 Comparaison par rapport aux attentes du monde informatique

Notre approche est en partie implémentée dans un outil mais reste encore théorique. Elle s'inscrit dans des travaux de recherche menés par Atos sur la vérification de code à partir de méthodes formelles et le prototype que nous avons implémenté ne possède pas aujourd'hui de communauté d'utilisateurs. Cependant, nous pouvons légitimement nous demander quelle est l'opinion de la communauté informatique sur ce type d'approche et plus généralement sur les méthodes formelles. Nous avons donc réalisé une enquête sur l'utilisation des méthodes formelles pour la vérification de programmes auprès d'un échantillon de cette communauté.

6.3.1 La population des personnes interrogées

Nous avons ciblé les personnes proches du génie logiciel avec les membres du GDR-GPL [26] ainsi que les personnes proches du domaine des systèmes embarqués avec les membres du DAS [27] Systèmes Embarqués et du CISEC [28]. L'enquête s'est déroulée entre Octobre 2013 et Janvier 2014 via un questionnaire électronique anonyme composé d'un maximum de 14 questions et diffusé par courriel. Le questionnaire est disponible en Annexe C.

Nous avons pu recueillir les réponses de 75 personnes. Parmi elles, nous comptons des industriels, des académiques mais également des étudiants. Les niveaux d'étude des différentes personnes interrogées montrent une forte majorité de personnes détenant un doctorat, comme présenté Figure 6.4. Les résultats montrent également que 82% des personnes interrogées connaissent des méthodes formelles appliquées à la vérification de programme.

6.3.2 L'expérience des personnes interrogées sur les méthodes formelles pour la vérification de programmes

Les questions de cette partie concernent uniquement le groupe de personnes qui ont répondu connaître des méthodes formelles appliquées à la vérification de programmes, soit 66 des personnes interrogées sur 75. Dans cette partie, les résultats de l'enquête montrent que ce groupe peut être séparé en deux catégories : les personnes qui ont déjà

26. Groupement de Recherche Génie de la Programmation et du Logiciel, `http://gdr-gpl.cnrs.fr/`
27. Domaine d'Activité Stratégique.
28. Club Inter-association des Systèmes Embarqués Critiques, `asso-cisec.org`

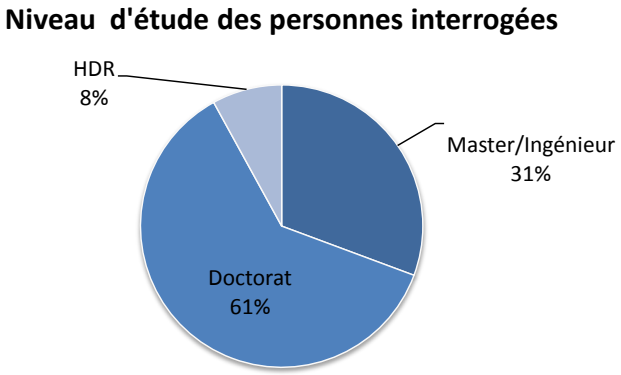


FIGURE 6.4 – Niveau de formation des personnes interrogées

utilisé les méthodes formelles dans le cadre de la vérification de programme, soit 82% du groupe, et ceux qui ne l'ont pas fait.

La difficulté principalement rencontrée lors de leur utilisation

L'enquête pose tout d'abord le problème de la principale difficulté rencontrée par les utilisateurs dans l'usage des méthodes formelles pour la vérification de programmes. La question offrait cinq possibilités : exprimer et formaliser les propriétés, comprendre les résultats fournis, il n'y a pas de difficulté, sans opinion ou autres (réponse libre).

L'enquête montre que la difficulté principalement rencontrée est l'expression et la formalisation des propriétés à vérifier avec environ 70% de réponses positives : 65% pour les personnes ayant déjà une expérience des méthodes formelles pour la vérification de programme et 92% pour les autres (Figure 6.5). La seconde difficulté rencontrée concerne l'interprétation des résultats fournis par la vérification. Les personnes interrogées avaient également la possibilité de fournir une réponse libre. Sept personnes, soit environ 11% du groupe, ont donné leurs propres réponses. Ces réponses étant toutes uniques, aucun résultat significatif ne s'en dégage [29]. Un récapitulatif des taux de réponses est donné.

L'avantage de l'utilisation des méthodes formelles dans le cadre de la vérification de programmes

L'enquête porte ensuite sur l'opinion des utilisateurs sur le fait que l'utilisation des méthodes formelles apporte un avantage à la vérification de programmes [30].

L'enquête montre que 62% des personnes interrogées possédant une connaissance des méthodes formelles sont totalement d'accord avec le fait que l'utilisation des méthodes

29. A titre d'exemple, nous pouvons citer les réponses suivantes : explosion combinatoire, passage à l'échelle, maîtrise industrielle de la méthode employée, etc.

30. Cette question donne le choix entre quatre niveaux d'opinion, 1 représentant que la personne interrogée n'est pas d'accord avec cette idée et 4 représentant l'extême opposé.

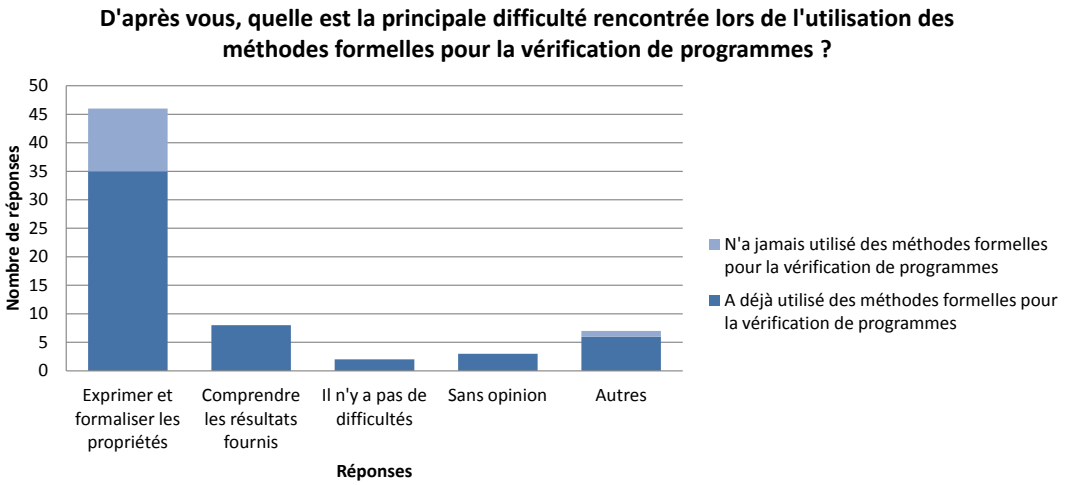


FIGURE 6.5 – Difficultés rencontrées lors de l'usage des méthodes formelles pour la vérification de programmes

formelles apporte un avantage à la vérification de programme et 33% de plus sont plutôt d'accord, ce qui donne 95% des personnes interrogées qui ont une opinion favorable à la question (Figure 6.6). Remarquons que l'ensemble des personnes interrogées ayant une expérience préalable des méthodes formelles pour la vérification de programmes pensent que ces méthodes sont avantageuses.

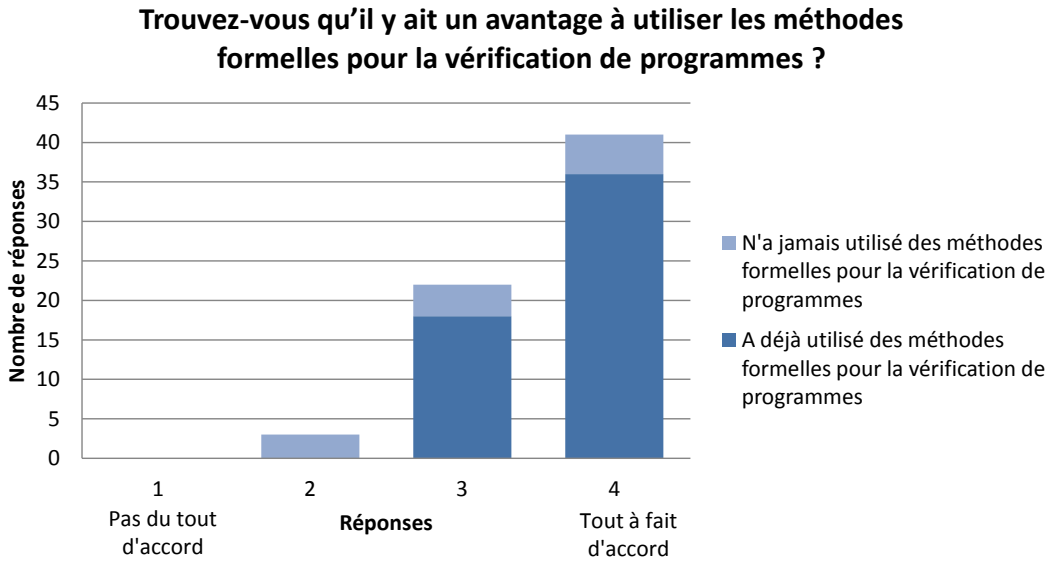


FIGURE 6.6 – Opinion des personnes interrogées ayant une connaissance des méthodes formelles pour la vérification de programmes sur l'avantage de leur utilisation

Le niveau d'investissement en formation attendu pour l'utilisation des méthodes formelles

L'enquête demande ensuite l'opinion des personnes interrogées sur l'investissement nécessaire en formation pour l'usage des méthodes formelles [31].

Les résultats de l'enquête montrent que 89% des personnes interrogées possédant une connaissance des méthodes formelles pensent qu'il y a un investissement non négligeable en formation à faire et 53% d'entre eux pensent que beaucoup d'investissement est nécessaire (Figure 6.7).

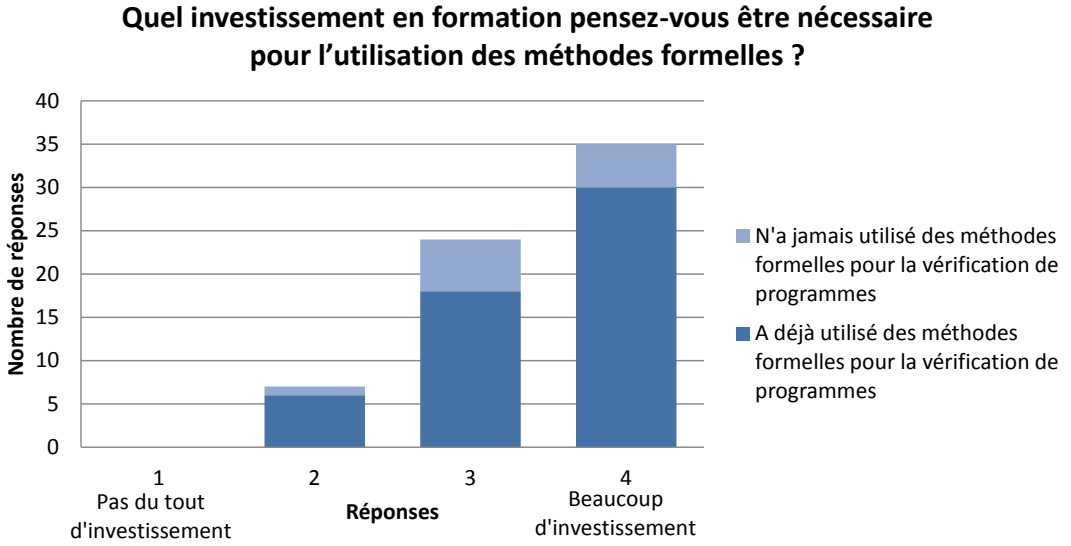


FIGURE 6.7 – Opinion des personnes interrogées ayant une connaissance des méthodes formelles pour la vérification de programmes sur l'investissement nécessaire en formation pour l'usage des méthodes formelles

6.3.3 Les attentes des personnes interrogées sur les méthodes formelles pour la vérification de programmes

Cette dernière partie de l'enquête s'adresse à toutes les personnes interrogées, qu'elles aient une connaissance ou non des méthodes formelles.

La place des méthodes formelles pour la vérification de programmes dans les processus de développement

Dans un premier temps, l'enquête cherche à connaître l'opinion des personnes interrogées sur l'utilisation des méthodes formelles qu'ils font ou qu'ils seraient prêt à

31. Quatre niveaux d'opinion possibles, 1 représentant que l'interrogé pense qu'aucun investissement n'est nécessaire et 4 représentant l'extrême opposé.

faire dans le cadre de la vérification de programmes. Dans ce cadre, l'enquête cherche à savoir si les personnes interrogées sont intéressées par leur usage en remplacement des techniques de vérification de programmes actuelles ou en ajoutant une nouvelle étape de vérification.

Les résultats de l'enquête montrent que 55% des personnes interrogées remplacent ou sont prêts à remplacer les techniques de vérification de programmes actuelles par un usage des méthodes formelles. Ils montrent également que 76% ajoutent ou sont prêts à ajouter une nouvelle étape de vérification à l'aide des méthodes formelles.

La légitimité des résultats obtenus par utilisation des méthodes formelles par un non expert

L'enquête se concentre ensuite sur le niveau de connaissance requis pour pouvoir exploiter les résultats fournis par les méthodes formelles. Nous voulons savoir si les personnes interrogées estiment que dans le cadre d'une vérification de programmes, un utilisateur peut utiliser les résultats donnés par un outil de preuve sans en connaître les fondements théoriques i.e. l'outil donne un résultat de vérification en masquant les détails de l'analyse formelle [32].

Les résultats sont partagés : 44% des personnes interrogées sont plutôt contre cette possibilité et 56% sont plutôt pour (Figure 6.8). Nous remarquons cependant que ce sont les personnes interrogées n'ayant pas d'expérience des méthodes formelles pour la vérification de programmes ou pas d'expérience dans ce domaine qui sont majoritairement contre (les 2/3).

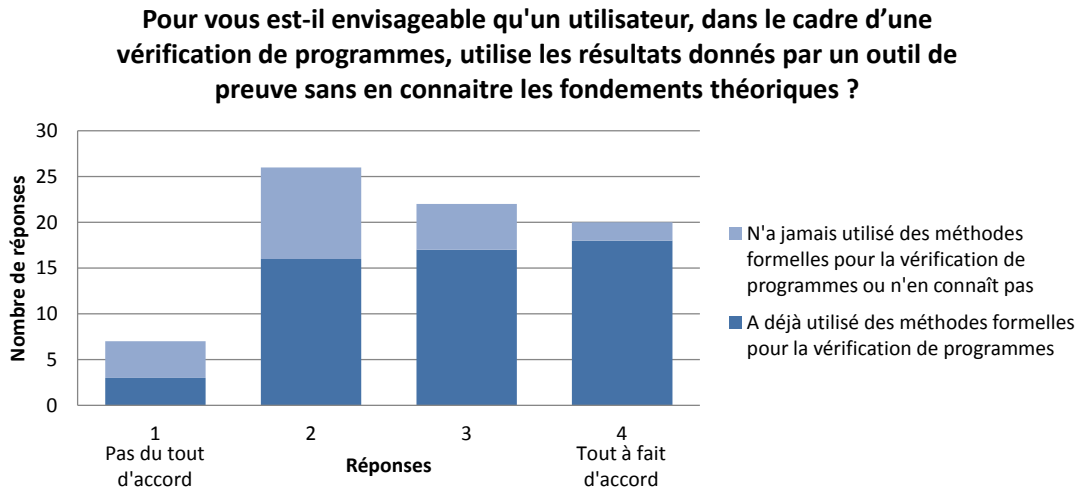


FIGURE 6.8 – Opinion des personnes interrogées sur l'usage des résultats donnés par un outil de preuve sans en connaitre ses fondements théoriques

32. L'enquête donne le choix entre quatre niveaux d'opinion, 1 représentant que la personne interrogée n'est pas du tout d'accord avec cette proposition et 4 représentant l'extrême opposé.

Nous souhaitons ensuite savoir si les personnes interrogées estiment nécessaire que l'utilisateur soit un expert des méthodes formelles employées pour légitimer les résultats obtenus de la vérification de programmes [33].

Les résultats sont également ici peu probants : 57% des interrogés sont plutôt contre cette proposition et 43% sont plutôt pour (Figure 6.9).

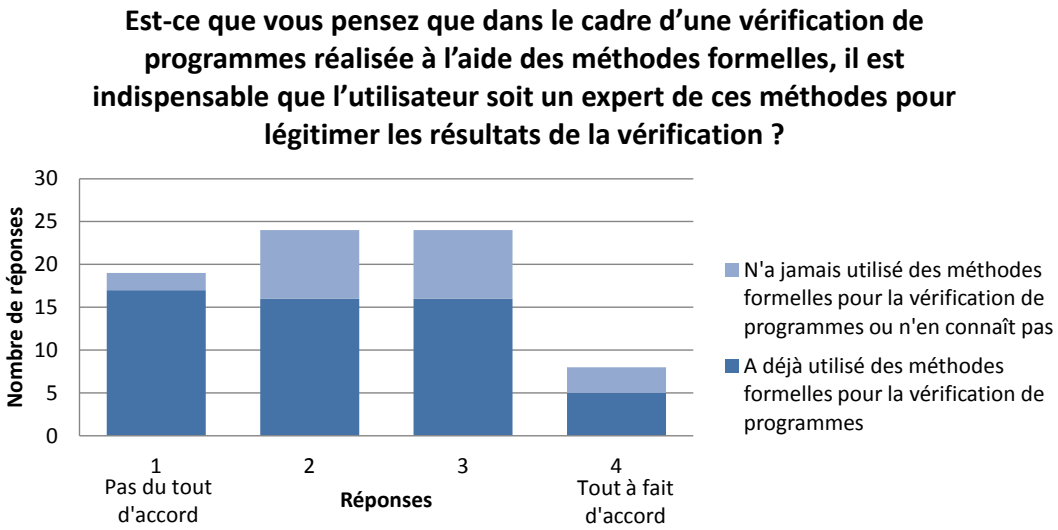


FIGURE 6.9 – Opinion des personnes interrogées sur la nécessité d'être un expert pour légitimer les résultats obtenus pour la vérification de programmes en utilisant des méthodes formelles

L'utilisation d'un outil automatique de vérification de programmes à l'aide des méthodes formelles

Pour finir, l'enquête s'intéresse à la mise en place d'un outil de vérification de programmes automatique à l'aide des méthodes formelles et son accueil par la population ciblée. Les personnes interrogées doivent répondre s'ils utiliseraient un tel outil s'il leur était proposé [34].

57% des interrogés sont convaincus qu'ils l'utiliseraient et 32% de plus sont également plutôt pour son utilisation (Figure 6.10).

33. quatre niveaux d'opinion, 1 représentant que la personne interrogée n'est pas du tout d'accord avec cette proposition et 4 représentant l'extrême opposé.

34. La question donne le choix entre quatre niveaux d'opinion, 1 représentant que la personne interrogée ne l'utiliserait pas et 4 représentant son contraire.

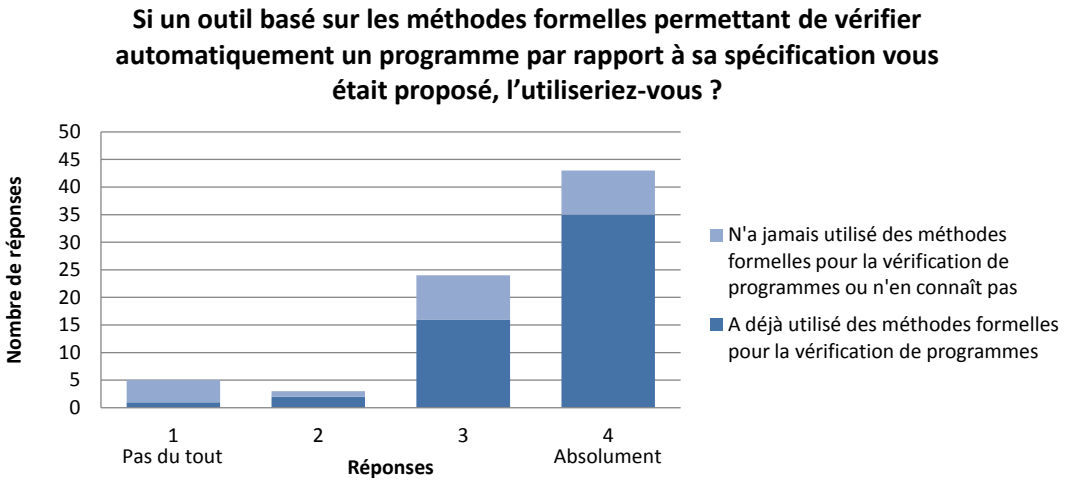


FIGURE 6.10 – Opinion des personnes interrogées sur l'utilisation d'un outil de vérification de programmes automatique basé sur les méthodes formelles

6.3.4 Conclusion sur l'enquête sur l'utilisation des méthodes formelles pour la vérification de programme

L'approche que nous avons définie tend à donner accès à l'utilisateur à la preuve déductive de code par une génération automatique des propriétés à vérifier, et ce quel que soit son niveau d'expertise sur les méthodes formelles. L'enquête que nous avons menée montre que la population interrogée, et possédant des connaissances sur les méthodes formelles, est consciente de la difficulté d'exprimer les propriétés à vérifier sur un programme et du fort coût d'investissement en formation pour se former à ces méthodes, comme nous le montre les résultats de l'enquête Figure 6.5 et Figure 6.7. Nous voyons aussi que la population interrogée est également très réceptive à la création d'un outil permettant la vérification automatique de programmes basée sur les méthodes formelles, comme nous le montre la Figure 6.10.

Si nous croisons les résultats obtenus pour la question sur la difficulté principalement rencontrée lors de l'usage des méthodes formelles pour la vérification de programmes et les résultats obtenus pour la question sur l'utilisation d'un outil permettant la vérification automatique de programmes basée sur ces même méthodes, nous pouvons observer Figure 6.11 une forte corrélation entre les personnes ayant des difficultés à exprimer et à formaliser les propriétés à vérifier et les personnes susceptibles d'utiliser un outil automatisé pour la vérification de programme. Cette observation nous permet d'être confiant sur l'intérêt de notre approche, aussi bien pour l'industriel Atos, que pour une population plus large du monde informatique.

Cependant, l'outil AGrUM que nous proposons n'est pas un outil magique. Une fois que l'approche et que l'implémentation seront arrivées à une maturité industrielle satisfaisante, nous pouvons certes espérer qu'elles permettront l'amélioration d'une partie

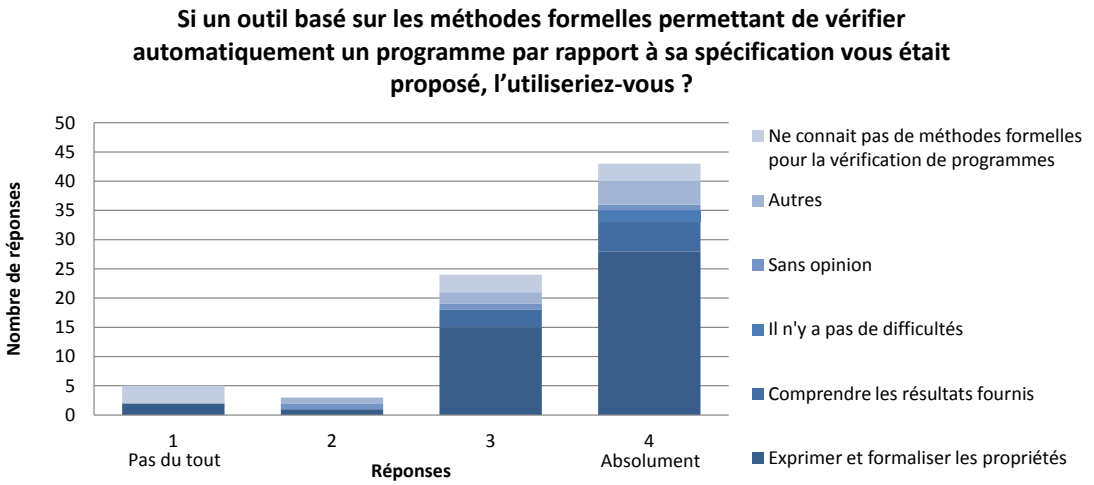


FIGURE 6.11 – Réponses des personnes interrogées sur l'utilisation d'un outil de vérification de programmes automatique basé sur les méthodes formelles en fonction de leurs réponses sur les difficultés rencontrées dans leur usage

des processus de vérification et offriront un environnement simple pour la vérification formelle à des non experts. Mais les opinions sont partagées concernant l'utilisation des méthodes formelles dans les tâches de vérification par des non experts, comme nous le montrent les résultats de l'enquête sur les Figures 6.8 et 6.9. Il semble nécessaire que l'utilisateur acquiert un minimum de connaissances sur les fondements de l'approche automatisée pour légitimer son utilisation. La mise au point de notre approche n'est pour l'instant qu'un premier pas vers une automatisation complète des méthodes formelles.

6.4 Comparaison par rapport aux travaux connexes

Nos travaux ne sont pas les seuls à ouvrir cette voie. D'autres ont exploré la génération automatique d'annotations à partir d'un modèle afin d'aboutir à une vérification formelle de la conformité d'un code source par rapport à sa conception.

[41] propose un moyen d'annoter automatiquement un code C selon une spécification composée d'automates SAM[35], en vue de sa vérification comportementale. SAM est un langage dédié pour la représentation comportementale de composants avioniques. L'article présente un algorithme pour générer ces annotations à partir du modèle. Les annotations peuvent être exprimées sous forme de contrats de fonction en ACSL pour une vérification via Frama-C ou dans un langage propre à l'outil de vérification CAVEAT. Si l'approche est similaire à la notre, les automates SAM qui sont utilisés dans l'approche sont différents de nos machines à états. La notion d'événement n'existe pas dans les automates SAM. De plus, les actions réalisables par ces automates SAM, qui sont la

35. Structured Automata Model.

production de valeurs de sortie, sont définies sur les transitions et non dans les états.

Le greffon Frama-C Aorai [36] permet de générer des annotations ACSL, à partir d'une spécification d'automates décrite soit dans le langage YA propre à l'outil, soit sous la forme d'une formule LTL (Logique Temporelle Linéaire). Aorai annote automatiquement le code source ciblé et la vérification peut ensuite être effectuée à partir des solveurs disponibles via Frama-C. Concrètement, la spécification sous forme d'automates nécessaire à Aorai représente une séquence d'appels et de retours de fonctions. Chacun d'entre eux peut être couplé à des propriétés portant sur des variables du programme. Si les annotations sont vérifiées, alors le greffon permet de s'assurer que le programme respecte sa spécification. Aorai se focalise sur l'appel de fonctions au niveau du programme alors que notre approche s'intéresse plus particulièrement aux comportements de ces fonctions. De plus, pour des utilisateurs non experts, une spécification textuelle réalisée en LTL ou en YA est beaucoup moins intuitive qu'une spécification réalisée à partir d'un diagramme de machine à états.

Dans [38], les auteurs présentent une méthode expérimentale pour la vérification automatique d'un générateur de code pour systèmes embarqués critiques. Cette méthode s'appuie sur la vérification par analyse statique de code C généré par rapport à une spécification sous forme de modèle au format Simulink [37]. Les auteurs définissent une spécification formelle des entrées/sorties de chaque bloc Simulink représentable. A partir de cette spécification formelle, ils proposent une génération automatique d'annotations ACSL sur le code C généré. La vérification est ensuite menée dans Frama-C. L'idée est sensiblement proche de la notre mais diffère sur le type de modèle d'entrée et les objectifs de vérification. L'objectif est ici de vérifier le générateur de code, il n'envisage pas que le code puisse être écrit par un opérateur humain à partir de la spécification.

Dans [80], les auteurs présentent les fondements théoriques d'un outil pour la génération d'annotations sur une implémentation logicielle à partir de propriétés issues de la théorie du contrôle et exprimées sur la conception du système de contrôle. Le but est d'obtenir un générateur de code prouvé. Le langage de conception est Scilab [38], une version open-source du langage Matlab [39]. L'implémentation est exprimée dans le langage C. Les propriétés sont transformées vers des annotations ACSL sur le code. Les annotations ACSL sont ensuite vérifiées par Frama-C. Les auteurs présentent deux méthodes. La première est une traduction directe des propriétés définies dans la conception et des opérateurs Scilab vers des annotations ACSL sur le code. La seconde se base sur un langage pivot nommé Lustre [69] pour la transformation en annotations ACSL afin de pouvoir prendre en compte différents langages pour la conception. La philosophie de l'approche est proche de la nôtre. Elle diffère par le type de système vérifié, par le langage de conception et les propriétés à vérifier.

[78] propose de générer des annotations JML [40] à partir d'une machine à états UML en vue de la vérification d'une implémentation Java. JML est un langage de spécification

36. `frama-c.com/aorai.html`
37. `www.mathworks.fr/products/simulink/`
38. `www.scilab.org`
39. `www.mathworks.fr/products/matlab/`
40. Java Modeling Language.

très similaire à ACSL. Il est cependant dédié au langage Java. Les auteurs présentent un outil pour la génération automatique de ces annotations. Ils se basent sur l'outil ArgoUML [41] pour la représentation des machines à états UML et se limitent à la vérification d'applications Java Card [42]. Le code source annoté est ensuite vérifié par l'outil ESC/-Java [58]. L'objectif de ces travaux est similaire au nôtre, s'assurer que le programme implémente le modèle de conception. Cependant, les contrats de fonction sont partiels, ils ne vérifient qu'une partie de la complétude de l'implémentation des transitions.

41. `argouml.tigris.org`

42. Java Card est un environnement logiciel pour cartes à puce permettant l'exécution d'un sous-ensemble Java.

Chapitre 7

Perspectives et conclusion

Les travaux que nous avons présentés dans cette thèse permettent d'apporter une première réponse à l'amélioration des processus de vérification de programmes par combinaison des méthodes formelles avec l'IDM. Ces travaux pourront être étendus suivant différentes perspectives dans la continuité de la thèse. Dans ce cadre, nous présentons les pistes que nous préconisons dans la Section 7.1. Enfin, nous concluons cette thèse dans la Section 7.2.

7.1 Perspectives de la thèse

7.1.1 Interprétation automatique des résultats

Identification des erreurs d'implémentation

Dans notre approche, la preuve déductive permet de vérifier les annotations représentant les propriétés à prouver sur l'implémentation et d'indiquer à l'utilisateur si celles-ci sont valides ou si la preuve n'a pas pu aboutir. Actuellement, il est à la charge de l'utilisateur d'interpréter ces résultats. Nous avons vu dans l'enquête décrite Section 6.3 que l'interprétation des résultats est la seconde difficulté la plus rencontrée lors de l'usage des méthodes formelles pour la vérification de programmes (voir Figure 6.5). Par conséquent, si nous voulons proposer industriellement notre approche à des non experts, il nécessaire de s'intéresser à l'interprétation des résultats renvoyés par la preuve déductive réalisée par Frama-C. Une première piste sera d'étudier l'interprétation des propriétés non validées et le renvoi à l'utilisateur d'indications adaptées sur les erreurs commises sur le code.

Cette piste a fait l'objet de travaux préliminaires qui se basent sur les travaux réalisés à l'ONERA par Claudine Chamontin au cours de son stage [25]. Si nous reprenons les cinq propriétés à vérifier pour les fonctions de transition que nous avons définies dans la Section 5.3, nous avons :

(a) pour l'état courant, si une transition sortante peut être déclenchée alors la fonction de transition renvoie l'état cible de cette transition ;

(b) la fonction de transition est sans effet de bord ;

(c) pour l'état courant, si une transition a été déclenchée alors sa garde doit-être vraie ;

(d) pour l'état courant, si aucune garde d'aucune transition sortante n'est vraie, alors aucune transition n'est déclenchée i.e. la fonction de transition renvoie $\emptyset$;

(e) si un état n'est pas pris en compte par une fonction de transition (aucune transition sortante de cet état n'est déclenchée par l'événement géré par cette fonction), cette fonction de transition ne déclenche aucune transition pour cet état i.e. la fonction de transition renvoie $\emptyset$.

Ces propriétés sont définies dans des patrons d'annotations. Les quatre premières sont regroupées dans les *behavior* de chaque état (le lien avec chaque annotation est donné Listing 7.1). La dernière est définie entièrement dans un *behavior* particulier, rappelé Listing 7.2.

```
behavior <nom Etat> :
    assumes current_state == nom Etat;
    (b) assigns \nothing;
    (a) ensures <Garde de la transition sortante 1>
            ==> \result == <Etat cible de la transition sortante 1>;
    ⋮
    (a) ensures <Garde de la transition sortante N>
            ==> \result == <Etat cible de la transition sortante N>;
    (c) ensures \result == <Etat cible de la transition sortante 1>
            ==> <Garde la transition sortante 1>;
    ⋮
    (c) ensures \result == <Etat cible de la transition sortante N>
            ==> <Garde la transition sortante N>;
    (d) ensures (!<Garde la transition sortante 1>
          && ...
          && !<Garde la transition sortante N>)
                    ==> \result == Null;
```

Listing 7.1 – Patron des annotations pour les propriétés (a) (b) (c) et (d) pour un état

```
behavior OtherStates :
    assumes current_state != <Etat 1>
              && ...
                 && current_state != < Etat n>;
    assigns \nothing;
    ensures \result == Null;
```

Listing 7.2 – Patron des annotations pour la propriété (e)

Une non-validation des annotations de chaque propriété va permettre de conclure à des erreurs différentes présentes dans l'implémentation. Ces erreurs vont également dépendre des combinaisons des différents types d'annotations non valides. Nous présentons un aperçu des interprétations actuellement identifiées.

– pour les annotations correspondantes à la propriété (a), il peut s'agir de plusieurs types d'erreurs. Si au sein d'un même *behavior*, il y une annotation non valide de ce type, il s'agit d'un problème d'implémentation des transitions (erreur sur la condition ou sur l'état, ou oubli de la transition). Si, toujours au sein d'un même *behavior*, toutes les annotations de ce type sont non valides, il peut s'agir d'un oubli de l'implémentation des transitions pour l'état correspondant au *behavior* ;

– pour les annotations correspondantes à la propriété (b), cela correspond à l'intrusion dans l'implémentation d'une instruction modifiant une variable globale. S'il s'agit uniquement d'une annotation correspondant à un *behavior* spécifique, l'instruction se trouve dans le bloc de code correspondant à cet état. Si toutes les annotations pour cette propriété sont non valides pour tous les *behavior*, l'instruction se trouve hors du bloc *switch/case* de la fonction ;
– pour les annotations correspondant à la propriété (c), si une des transitions est non valide pour un *behavior*, il peut s'agir de l'ajout d'une transition vers un état déjà ciblé par le *behavior* ;
– pour les annotations correspondantes à la propriété (d), il s'agit de l'ajout d'une transition non existante dans la conception vers un état non accessible à partir de l'état source testé. Dans le cas où toutes les annotations pour cette propriété sont non valides pour tous les *behavior*, il peut s'agir d'une mauvaise initialisation de la variable de sortie. Enfin, dans le cas où toutes les annotations pour cette propriété sont non valides pour tous les *behavior*, ainsi que toutes les annotations pour les propriétés (a) et (c), il peut s'agir d'une modification non autorisée de la variable de sortie en fin de fonction ;
– pour les annotations correspondant à la propriété (e), il s'agit de l'ajout du traitement d'un état non pris en compte par cette fonction.

Notons que l'interprétation des résultats dépend fortement du patron d'implémentation employé. Une étude plus poussée devra être réalisée pour fournir à l'utilisateur des indications plus précises sur ces résultats. Comme réalisé dans [25], nous pourrons également ajouter des annotations pour affiner ces résultats, même si ces annotations ne sont pas nécessaires pour la preuve. Par exemple, l'utilisation d'une équivalence pour l'annotation représentant la propriété (d) permettrait d'interpréter avec plus de certitude l'oubli de l'implémentation d'une transition.

Il faut également noter que nous devrons prouver que notre approche couvre tous les cas d'erreurs d'implémentation possibles pour prouver que notre approche est valide.

Extension de l'approche outillée proposée

L'ébauche des possibilités d'interprétation des résultats renvoyés par Frama-C que nous venons de présenter pourra être implémentée dans la nouvelle version d'AGrUM, en nous basant sur un prototype réalisé par [25]. Nous pourrons ainsi proposer une extension de l'approche outillée présentée Figure 6.2 dans la Section 6.2.1. Cette extension est visible Figure 7.1.

7.1.2 Preuve complète de l'implémentation du comportement d'une machine à états

Nous avons montré comment vérifier formellement et automatiquement la conformité de l'implémentation des fonctions de transition d'une machine à états par rapport à sa conception dans la Section 5.3. Nous avons également présenté une solution partielle pour la vérification automatique de l'implémentation de la fonction d'activité en Annexe B.

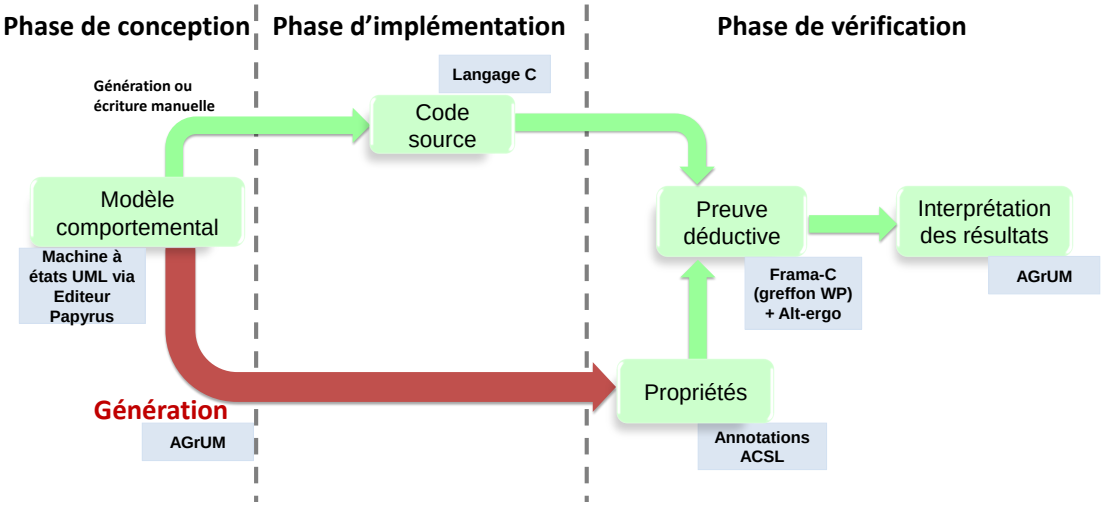


FIGURE 7.1 – Extension de notre approche outillée

Bien que ces fonctions représentent le comportement défini par l'utilisateur, elles ne représentent qu'une partie de l'implémentation d'une machine à états. Leur vérification formelle ne permet donc d'aboutir qu'à une preuve partielle de l'implémentation du comportement d'une machine à états. Pour aller vers une preuve complète, il faudra vérifier les fonctions restantes : `RTC_tick`, `RTC_c`, `Cycle` et `Cycle_0`. A la comparaison des fonctions de transition et d'activité, ces fonctions décrivent le fonctionnement d'une machine à états. Elles sont donc identiques pour chaque machine à états et le concepteur n'a aucune influence sur leur implémentation. Pour réaliser leur preuve, il suffira de définir leurs contrats une seule fois afin de pouvoir l'appliquer à toutes nos machines à états. Leur structure ne dépendant pas du comportement décrit dans le modèle, ils pourraient ensuite être incorporés dans le code à partir d'un patron défini dans l'outil AGrUM.

La définition des contrats pour ces fonctions est cependant plus complexe que pour les fonctions `T_tick`, `T_c` et `A`. Les fonctions `RTC_tick` et `RTC_c` se basent sur des appels aux fonctions `T_tick`, `T_c` et `A` pour leur implémentation. Les fonctions `Cycle` et `Cycle_0` se basent ensuite sur des appels aux fonctions `RTC_tick` et `RTC_c` pour leur implémentation. Un graphe d'appel des fonctions du programme est décrit Figure 7.2.

Nous pourrons envisager de nous baser sur l'outil Aorai (voir Section 6.4) pour une génération automatique de contrats ACSL afin de vérifier le flot d'appel des différentes fonctions. Cependant, ces contrats générés ne permettent pas de vérifier l'intégrité des données qui sont transmises entre les différents appels. Par exemple, ils ne permettent pas de détecter l'ajout d'une instruction modifiant une variable entre deux appels de fonctions. L'emploi de cette méthode implique donc de définir des contrats supplémentaires pour vérifier complètement le comportement. Pour vérifier ce flot de données, nous pourrons réfléchir à l'emploi de la technique présentée dans l'Annexe B pour la vérification du comportement de la fonction d'activité `A`, qui s'appuie sur des observateurs et

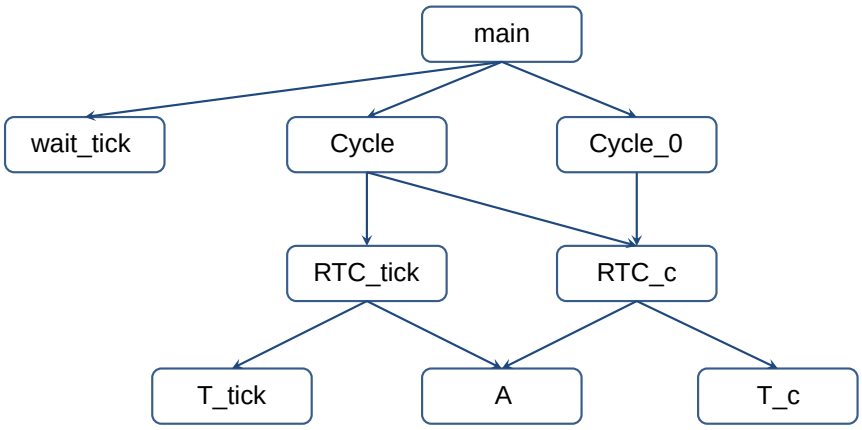


FIGURE 7.2 – Graphe d'appel des fonctions de nos machines à états

des bouchons sur les fonctions appelées. Cependant, notons que nous possédons déjà les contrats décrivant les fonctions appelées (`T_tick`, `T_c` et `A`). Il faudra donc réfléchir à un moyen de les exploiter pour la preuve des fonctions appelantes. De plus, les fonctions `Cycle` et `Cycle_0` contiennent des boucles. Il faudra, en plus de définir les contrats permettant de prouver les propriétés désirées, générer des variants pour aboutir à la preuve et des invariants pour vérifier la terminaison.

Il restera ensuite à vérifier la fonction principale décrivant le fonctionnement global (le `main`) et la fonction `wait_tick` avec des méthodes similaires.

7.1.3 De la preuve du code source vers la preuve de code exécutable

Notre approche tend à proposer une étape à forte valeur ajoutée pour la vérification de programmes. Nous pouvons envisager d'étendre notre approche à la vérification du code exécutable en remplacement des tests, même si nous avons vu dans la Section 6.3, que l'utilisation des méthodes formelles dans ce contexte était sujette à discussion.

Pour pouvoir prétendre à étendre notre approche, il faudra tout d'abord s'assurer que le code ne présente pas d'erreur d'exécution [1]. Dans le cadre de l'utilisation de l'outil Frama-C pour la vérification, le greffon Value Analysis [2] et le greffon RTE [75] permettent ce type de vérification. L'avantage du greffon RTE est d'être automatiquement combinable au greffon WP que nous utilisons. Il permet de générer automatiquement des annotations pour vérifier des erreurs d'exécution telles que les divisions par zéro, les débordements d'entiers (signés ou non-signés) ou encore des accès mémoire invalides. Pour une utilisation future, il faudra étudier les avantages de ces deux greffons et choisir le plus adapté à notre approche.

Comme nous l'avons préalablement mentionné dans la Section 6.2.2, il faudra ensuite s'intéresser à la préservation des propriétés prouvées au niveau du code source dans

1. En anglais : runtime error.
2. `frama-c.com/value.html`

le code exécutable, conformément aux attentes du standard de certification DO-333. Dans ce cadre, le projet CompCert [3] étudie la vérification de compilateurs pour des logiciels embarqués critiques. L'outil qui en résulte est le compilateur du même nom, CompCert [85, 86], qui garantit que le code exécutable qui est produit se comporte tel que spécifié par la sémantique du code source exprimé en C. La vérification du compilateur est réalisée formellement à l'aide de l'assistant de preuve Coq. Nous pourrons donc réfléchir à l'utilisation de ce compilateur en complément de notre approche afin d'obtenir une chaîne complète de vérification permettant de prouver du code exécutable par rapport à son modèle de conception.

7.1.4 Prise en compte d'autres patrons d'implémentation de machines à états

Le patron d'implémentation que nous avons défini est propre à nos machines à états. La décomposition en différentes fonctions est directement liée à la sémantique formelle que nous avons définie mais l'implémentation des fonctions de transition s'appuie sur un patron d'implémentation courant basé sur des *switch/case*. Il existe d'autres patrons d'implémentation plus complexes utilisés dans l'industrie.

A titre d'exemple, [116] présente différents patrons d'implémentation des machines à états UML en C et C++. Il y fait d'ailleurs référence à un patron proche du notre pour le choix des transitions, le patron *Nested Switch Statement* [4]. Comme son nom l'indique, ce patron se base sur l'utilisation de plusieurs *switch/case*, un pour le choix de l'état courant et un autre pour le choix de l'événement pour chaque état possible. L'état ciblé et les actions à réaliser sont ensuite choisis par une succession de *if/else if* portant sur les gardes des transitions.

L'auteur décrit également d'autres patrons d'implémentation classiques tels que les patrons de type *state design pattern* [5] et *stable table pattern* [6]. Le *state design pattern* consiste à représenter chaque état par une classe. Le choix des transitions est implémenté dans les méthodes de ces classes, une méthode pour chaque événement possible. Ce patron se base sur le concept de polymorphisme et est donc utilisé pour des implémentations en langage orienté objet tel que le C++. Le *stable table pattern* consiste à représenter le choix des transitions sous forme d'un tableau. Il en existe plusieurs variantes. La plus courante se base sur un tableau à deux dimensions, les lignes représentant les états et les colonnes représentant les événements. Les cellules représentent alors les actions de la transition et son état d'arrivée. Dans cette variante, il n'y a pas de référence aux gardes des transitions. D'autres variantes permettent d'inclure ces gardes en définissant des fonctions de transition dans chaque cellule du tableau. Pour plus de détails sur ces patrons d'implémentation, nous invitons le lecteur à consulter l'ouvrage [116].

Nous pourrons donc réfléchir à la possibilité d'adapter certains des patrons d'implémentation existants à nos machines à états et d'adapter en conséquence nos contrats de

3. `compcert.inria.fr`
4. En français : Structures switch/case imbriquées.
5. En français : patron par état.
6. En français : patron par table d'états.

fonction actuels. Nous pourrons également envisager de proposer dans l'outil AGrUM un choix entre plusieurs patrons d'implémentation et donc différentes options de génération pour ces contrats.

7.1.5 La vérification de modèles

Notre approche nécessite un modèle de conception correct en entrée afin de pouvoir vérifier un code correct, tel que nous l'avons décrit dans la Section 5.5.

Comme le souligne notre enquête sur les retours d'expérience, Section 4.2.1 et 4.2.2, la vérification de modèle est un point important pour les utilisateurs. Même si dans ce travail, nous ne nous sommes pas intéressés à la vérification de modèle, nous avons pris pour postulat que les modèles étaient corrects [7], il est cependant possible d'étendre notre approche avec une étape de vérification, étape réalisée en amont de notre processus.

Pour permettre une vérification automatique des modèles d'entrée dans notre approche, deux pistes de recherche sont envisageables :

- définir complètement le méta-modèle de notre sous-ensemble UML ;
- nous baser sur le méta-modèle UML et définir des contraintes OCL qui vont représenter les restrictions apportées par notre sous-ensemble ;

Notons ici que la seconde piste est plus proche de la philosophie que nous avons suivi tout au long de ces travaux : rester compatible avec le standard UML. En effet, utiliser des contraintes OCL permettrait de vérifier la restriction d'UML que nous faisons sans introduire un nouveau langage à proprement parler. Il pourrait être également possible de vérifier, pour la machine à états modélisée, qu'il existe, au cours de chaque cycle, au moins un état où plus aucune transition soumise au *completion event* ne peut être déclenchée (comme nous l'avons défini Section 4.3.2). L'utilisation de propriétés temporelles discrètes est alors une piste envisageable pour la vérification de cette restriction particulière.

Nous pourrons également réfléchir à un moyen pour l'utilisateur de vérifier le comportement qu'il spécifie, à travers l'écriture de ses propres propriétés ou bien par simulation du modèle.

7.1.6 Evolution de notre sous-ensemble UML

Le sous-ensemble des machines à états UML que nous utilisons est limité à un ensemble basique de concepts. Pour de futurs travaux, nous pourrons le faire évoluer sur trois axes principaux.

Prise en compte du multi-événements

Notre sous ensemble UML est limité à l'utilisation de deux événements : le *completion event* et l'événement *tick*, qui est le seul événement externe autorisé dans nos machines à

7. Dans les faits nous avons implémenté dans AGrUM une partie des opérations de vérification (AGrUM vérifie que le modèle respecte une partie des restrictions syntaxiques de notre sous-ensemble UML).

états. Ce choix est historique, puisque nous nous sommes basés sur les besoins exprimés pour un sous-ensemble UML dédié à la spécification logicielle (voir Section 4.1). Nous pourrons étudier une extension de notre sous-ensemble à la prise en compte de différents événements externes afin de pouvoir toucher plus de cas de modélisation avec notre approche.

Dans ce cadre, nous avons commencé à explorer une solution. Concernant la sémantique du sous-ensemble UML employé, elle consisterait à remplacer la fonction T_{Tick} par une fonction T_{ext} de la forme $T_{ext} : S \times E_{ext} \times V \rightarrow S \cup \{\emptyset\}$ où E_{ext} représente l'ensemble des événements externes. Les fonctions T_c, rtc et $Cycle$ conserveraient leur fonctionnement (rtc_{tick} serait simplement renommé rtc_{ext}). La fonction $Cycle$ ne représenterait plus alors un cycle d'un top d'horloge, mais un cycle de traitement d'événement (le traitement de l'événement externe courant puis tous les *completion event* qui en découlent). Ainsi, nous conserverions la majeure partie de notre sémantique formelle.

Concernant l'implémentation, nous aurions tout d'abord l'ajout d'un type énuméré pour les événements. Puis, seul le patron de `T_tick` serait remplacé par le patron de `T_ext`. Cette fonction serait représentée par des *switch/case* imbriqués : un premier pour le choix des états et puis un pour chaque état et qui porterait sur le choix de l'événement. Un aperçu du nouveau graphe d'appel est donné Figure 7.3 . Notons que la fonction `dispatch_external_event` vient remplacer la fonction `wait_tick` qui permettait de simuler l'arrivée d'un top d'horloge. La fonction `dispatch_external_event` sera chargée de donner le prochain événement à traiter à la machine à états.

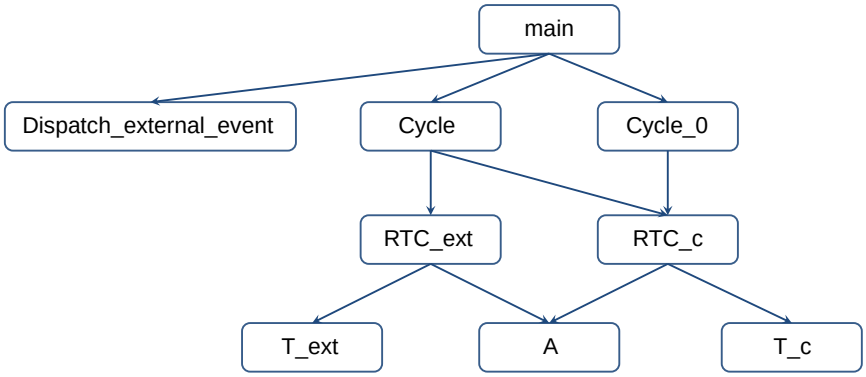


FIGURE 7.3 – Graphe d'appel des fonctions de nos machines à états pour l'extension au multi-événements

Concernant les propriétés à vérifier, les propriétés pour la fonction `T_ext` seraient raffinées et incorporeraient les événements dans leur définition. Les patrons de génération des contrats de fonction seraient également raffinés. Les événements interviendraient dans ces contrats au niveau de la définition des clauses *ensures* : les conditions de transition vers un état source seraient désormais exprimées sous la forme d'une conjonction d'un événement et de la garde de la transition. Les contrats des fonctions `T_c` et `A` resteraient inchangés.

Cette piste nécessite encore une formalisation de la nouvelle sémantique du sous-

ensemble UML, des nouveaux patrons d'implémentation et de génération d'annotations ainsi qu'une évaluation par des tests. Nous sommes cependant confiants quant à son efficacité.

Les machines à états hiérarchiques

Nous pourrons également étendre nos machines à états aux états hiérarchiques, tels que le *CompositeState* et le *SubmachineState* (voir leur description dans la Section 2.2.2). Leur utilisation permettrait de modéliser des systèmes plus complexes et d'obtenir une modélisation plus modulaire, en factorisant certains comportements ou en réutilisant d'autres comportements définis entièrement dans des machines à états.

Aujourd'hui, si nous voulions pouvoir traiter des machines à états hiérarchiques directement à partir de notre sous-ensemble et des patrons d'implémentation et de génération d'annotations que nous avons définis, nous pourrions envisager "d'aplatir" ces machines à états hiérarchiques pour obtenir une machine à états avec des états simples équivalente. Cependant, l'implémentation qui en résulterait risquerait d'être lourde et peu optimale. La prise en compte des états hiérarchiques ne pourra donc se faire rigoureusement qu'à partir d'une extension de notre sous-ensemble et de notre approche. De la même manière que pour la prise en compte de multiples événements externes, cette extension impliquera de définir un nouveau patron d'implémentation et par conséquent, de définir de nouveaux contrats de fonction pour la vérification à partir de nos travaux actuels.

Expression de la sémantique formelle du sous-ensemble UML en ACSL

La sémantique de notre sous-ensemble UML est actuellement définie à partir de l'arithmétique et de la logique du premier ordre multi-typée. Nous avons vu dans la Section 5.1 que le langage ACSL permettait la définition de spécifications logiques à partir de lemmes, d'axiomes, de prédicats et de fonctions logiques. Nous pourrons donc réfléchir à un moyen d'exprimer notre sémantique formelle à partir de ces concepts ACSL et de les réutiliser pour exprimer nos contrats de fonction. L'usage du même langage pour la définition de la sémantique et des annotations pourrait ainsi permettre de faciliter les vérifications par revue de la couverture de la conception par nos annotations.

7.2 Conclusion de la thèse

Les travaux menés dans cette thèse avaient pour objectif de répondre à la problématique portant sur l'amélioration des processus de vérification logicielle en combinant les méthodes formelles et l'IDM. L'axe de recherche qui avait été défini pour répondre à cette problématique était de proposer une approche adaptée à des utilisateurs non experts en automatisant la vérification formelle d'une partie du comportement de l'implémentation d'une machine à états UML par rapport à son modèle de conception.

Nous avons décrit, dans le Chapitre 3, le contexte industriel de cette thèse CIFRE qui a guidé la définition de notre axe de recherche. Nous avons ensuite défini dans le Chapitre 4 un sous-ensemble des machines à états UML propre à la conception d'un

logiciel embarqué pour conduire nos recherches, en nous basant sur des travaux industriels préliminaires se rapportant à la spécification logicielle. Nous avons également pu y présenter l'intérêt porté par les utilisateurs à la mise en place de techniques IDM pour les phases de spécification.

L'approche proposée pour répondre à notre problématique a été décrite dans le Chapitre 5 où nous avons défini un patron d'implémentation C pour nos machines à états UML, ainsi que les propriétés à vérifier et leurs patrons de génération sous forme d'annotations sur le code. Nous avons ainsi pu mettre en place une preuve automatique d'une partie de l'implémentation, dédiée aux fonctions de transition, en s'appuyant sur le contexte technique propre à l'industriel Atos.

Enfin, nous avons montré dans le Chapitre 6 que notre approche était automatisable pour des utilisateurs non experts en implémentant la génération des annotations dans un prototype se présentant sous la forme d'un greffon Eclipse qui peut s'intégrer dans une chaîne outillée comprenant un éditeur de modèles UML tel que Papyrus et un outil de preuve tel que Frama-C. Nous avons également montré que l'approche proposée permettait de répondre à certains objectifs de certification de la norme DO-333 du domaine avionique et qu'elle concordait avec une partie des attentes du monde informatique sur les méthodes formelles.

Ainsi, nos travaux permettent de proposer un processus de vérification logicielle se basant sur des techniques IDM et donnant à des utilisateurs non experts l'accès aux méthodes formelles et aux outils associés dans des développements pouvant nécessiter une certification. Ils ont également permis à l'industriel Atos de développer son savoir-faire et de disposer d'un premier démonstrateur. Bien que l'approche proposée se base sur le contexte technique propre à cet industriel, elle permet également de toucher une plus grande communauté par l'utilisation du standard UML. Elle devra être industrialisée afin de pouvoir évaluer précisément l'amélioration qu'elle peut apporter aux processus de vérification logicielle.

Annexes

Annexe A

Description du sous-ensemble formel du diagramme d'activité UML

Comme nous l'avons défini dans [42], ce sous-ensemble des activités UML permet la représentation de traitements synchrones, séquencés, non concurents et non récursifs.

Fondamentalement, une activité UML est composée d'un ensemble d'actions (*Action*), de nœuds de contrôle (*ControlNode*) et de nœuds de données (*ObjectNode*). Les actions et les nœuds de contrôle sont reliés entre eux par des flots de contrôle (*ControlFlow*) qui permettent de séquencer les actions. les nœuds de données sont reliés entre eux par des flots de données (*ObjectFLow*) représentant le transfert de données. Chaque action peut posséder des nœuds de données : ils représentent les données entrantes et sortantes de l'action.

Dans notre sous-ensemble, les actions sont limitées au *CallOperationAction* pour l'appel et l'exécution d'opérations externes stipulées comme synchrones, aux *ReadStructuralFeatureAction* et *AddStructuralFeatureValueAction* pour la gestion des variables globales, aux *CreateObjectAction*, *AddVariableValueAction* et *ReadVariableValueAction* pour la gestion de variables locales, à l'*OpaqueAction* pour la représentation d'une instruction C élémentaire et au *ValueSpecificationAction* pour la spécification de constantes. Les nœuds de contrôle sont limités aux *InitialNode*, *ActivityFinalNode*, *DecisionNode* et *MergeNode*. Les nœuds de données sont limités aux *InputPin* et *ValuePin* pour représenter les données entrantes d'une action, à l'*OutputPin* pour les données sortantes d'une action, et à l'*ActivityParameterNode* pour la représentation de données entrantes/sortantes de l'activité dans le cadre d'activités paramétrées. En complément de ces limitations, nous considérons qu'une action possède un unique flot de contrôle entrant et un unique flot de contrôle sortant. De même pour les nœuds de données avec les flots de données. Nous considérons également que le séquencement des actions est uniquement régi par les flots de contrôle. Enfin, nous interdisons les boucles dans le modèle.

Un exemple de diagramme d'activité est donné Figure A.1. Cette activité représente la vérification de la température d'un composant à partir d'un capteur placé sur celui-

ci. L'activité exécute tout d'abord l'opération `TMP_StartConv(inout sensor_id:integer)` pour demander un traitement de donnée au capteur à partir de son ID (`sensor_id`). Puis, l'activité exécute l'opération `TMP_GetTemp(in sensor_id:integer, return temp_value: integer):integer` pour récupérer cette valeur (`temp_value`) et fait un test sur celle-ci. Le résultat du test est stocké dans une variable globale (`Result`).

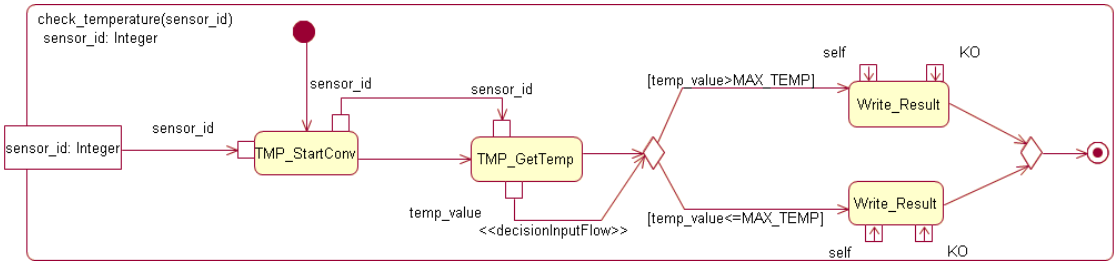


FIGURE A.1 – Exemple d'un diagramme d'activité

Annexe B

Solution pour la vérification de l'implémentation de la fonction d'activité de notre sous-ensemble UML

B.1 Méthode

B.1.1 Principe de la vérification des flots de données implémentés à partir d'un diagramme d'activité UML

Dans [42], nous avons présenté une solution pour vérifier le flot de données des activités UML au niveau de leur implémentation. Une activité UML est composée d'un flot de contrôle qui régit le séquencement des actions et un flot de donnée qui gère le passage de donnée d'action en action. Afin de s'assurer que ce flot de données est respecté dans l'implémentation entre chaque appel de fonction externe, il est nécessaire de vérifier que les données d'entrée et de sortie de ces fonctions correspondent à celles attendues (i.e. s'il existe un flot de donnée entre une fonction A vers une fonction B, la donnée d'entrée de la fonction B correspond à la donnée de sortie de la fonction A).

La méthode de vérification employée s'inspire d'une méthode basée sur des observateurs décrite dans le standard OCL pour la vérification de l'envoi et de la réception de messages paramétrés. Pour l'appliquer à de la vérification statique de code via des annotations ACSL, il a été nécessaire de définir trois types d'éléments pour la vérification :

- Un observateur : pour observer l'appel d'une fonction externe au sein d'une activité. Cet observateur est constitué d'un ensemble de contextes correspondant chacun aux valeurs d'entrée/sorties des paramètres de la fonction à chacun de ses appels. Un compteur est également défini pour compter et identifier les appels.
- Des stubs [1] : pour simuler chaque fonction externe et ainsi pouvoir peupler l'ob-

1. En français : bouchon.

servateur.
– Des contrats de vérification : pour vérifier les propriétés sur les flots de données à partir des données recueillies par l'observateur.

Cette approche permet de vérifier les flots de données entre différents appels de fonctions externes pour un sous-ensemble des activités UML. La preuve a pu aboutir sur des exemples composés de quelques dizaines de nœuds.

B.1.2 Application à l'appel d'activité

Dans la lignée de cette méthodologie, nous voulons vérifier l'appel d'activités au sein d'une machine à états. Pour ce faire, nous réutilisons les concepts de stub et d'observateurs décrits précédemment pour évaluer l'appel de ces activités.

Dans notre démarche, cet appel d'activités est représenté dans notre sémantique et sur notre code par une fonction d'activité A qui, suivant l'état courant de la machine à états, va appeler la fonction correspondante à l'activité de l'état. Si nous prenons la machine à états décrivant le comportement d'un train d'atterrissage automatisé Figure 4.10, un exemple de sa fonction d'activité implémentée en C est donné B.1.

```
void A(State current_state){
        switch(current_state)
        {
        case WaitingForTakeoff :
                start_timer();
                break;
        case StartRaisingGear :
                pump_on();
                break;
        case RaisingGear :
                dir_up();
                break;
        case GearUp :
                pump_off();
                break;
        case StartLoweringGear :
                pump_on();
                break;
        case LoweringGear :
                dir_down();
                break;
        case GearDown :
                pump_off();
                break;
        }
}
```

Listing B.1 – Implémentation de la fonction d'activité de la machine à états Figure 4.10

Pour la suite de l'annexe, nous appellerons *activité* la fonction représentant une activité dans le code et *fonction d'activité* la fonction de la machine à états permettant de savoir quelle activité exécuter suivant l'état courant.

L'observateur

L'observateur représente une série de contextes, un pour chaque appel de l'activité. Ce contexte est une structure C qui correspond aux valeurs des variables d'entrée/sortie modifiées par l'activité. Similairement à l'approche précédente, un compteur est également implémenté parallèlement à l'observateur afin de compter le nombre d'appels de cette activité au cours d'un même appel de la fonction d'activité.

Les patrons pour le contexte, l'observateur et le compteur pour l'activité `pump_off` de l'exemple du train d'atterissage sont donnés Listing B.2.

```
// Definition of pump_off_context
typedef struct {
        struct {
                  Pump_State pump_state ;
                  Pump_Dir pump_dir;
                } in;
        struct {
                  Pump_State pump_state ;
                  Pump_Dir pump_dir;
                } out;
} pump_off_context;


// Definition of pump_off_observer
/*@ ghost extern pump_off_context pump_off_observer[];*/


// Definition of pump_off_counter
/*@ ghost extern unsigned int pump_off_counter;*/
```

Listing B.2 – Contexte, observateur et compteur de l'activité `pump_off`

Le stub

Le fichier de stub va permettre de simuler le comportement de l'activité appelée. En effet, nous ne nous intéressons pas au comportement détaillé de l'activité. Nous souhaitons seulement savoir si elle a été appelée correctement par la fonction d'activité suivant l'état courant. Nous simulons donc l'activité afin de peupler l'observateur.

Cette simulation est réalisée grâce à un contrat de fonction ACSL décrivant le comportement attendu de l'activité appelée. Ce contrat va décrire comment est peuplé l'observateur. Lorsqu'une fonction n'a pas de corps mais possède un contrat de fonction, Frama-C va considérer que ce contrat de fonction est l'implémentation de la fonction (i.e. il ne prouve pas le contrat de fonction, il l'admet).

Un exemple de contrat de stub pour l'activité `pump_off` est donné Listing B.3.

```
/*@ assigns pump_state, pump_dir, pump_off_observer[pump_off_counter],
             pump_off_counter;
ensures \old(pump_state)
                == pump_off_observer[\old(pump_off_counter)].in.pump_state;
ensures \old(pump_dir)
                == pump_off_observer[\old(pump_off_counter)].in.pump_dir;
ensures pump_off_observer[\old(pump_off_counter)].out.pump_state
                == pump_state;
```

```
ensures pump_off_observer[\old(pump_off_counter)].out.pump_dir == pump_dir;
ensures pump_off_counter == \old(pump_off_counter) + 1 ;
*/
extern void pump_off(void);
```

Listing B.3 – Contrat de stub de l'activité `pump_off`

Le contrat de vérification

Nous allons chercher à vérifier que suivant l'état courant, la fonction d'activité appelle la bonne activité, et uniquement l'activité décrite dans la conception. Pour la fonction d'activité, nous allons donc vérifier plusieurs propriétés pour chaque état :

(a) les valeurs d'entrée des variables modifiées par l'activité ciblée correspondent aux valeurs d'entrée de ces variables à l'appel de la fonction d'activité ;

(b) les valeurs de sortie des variables modifiées par l'activité ciblée correspondent aux valeurs de sorties de ces variables après l'exécution de la fonction d'activité ;

(c) les valeurs des autres variables globales ne sont pas modifiées par la fonction d'activité ;

(d) une activité n'est appelée qu'une seule fois pour chaque état (i.e. le compteur d'appel ne dépasse jamais plus d'un appel).

Ces propriétés sont exprimées sous la forme de contrats ACSL sur la fonction d'activité, regroupés en *behavior* ACSL suivant chaque état. Chaque propriété correspond à une clause particulière de ce contrat.

- pour la propriété (a), en postcondition, nous pouvons vérifier que les valeurs d'entrée enregistrées dans l'observateur de l'activité appelée correspondent bien aux valeurs d'entrée de la fonction d'activité pour cet appel, pour chaque variable globale modifiable par l'activité appelée. Ceci est exprimable dans le contrat ACSL à partir d'une clause *ensures*.
- pour la propriété (b), de même, en postcondition, nous pouvons vérifier que les valeurs de sorties enregistrées dans l'observateur de l'activité appelée correspondent bien aux valeurs de sorties de la fonction d'activité pour cet appel, pour chaque variable globale modifiable par l'activité appelée. Ceci est exprimable dans le contrat ACSL à partir d'une clause *ensures*.
- pour la propriété (c), nous définissons une clause *assigns* pour stipuler les emplacements mémoires, et donc les variables qui vont être modifiées au cours de la fonction. Il est considéré que les variables qui ne sont pas déclarées dans cette clause restent inchangées.
- pour la propriété (d), il suffit de vérifier en postcondition que le compteur d'appel ne dépasse jamais 1 (ou 0 suivant d'où nous partons). Néanmoins, il n'est pas vraiment nécessaire d'expliciter cette propriété. Si l'activité est appelée plus d'une fois, les propriétés (a) ou (b) ne seront pas vérifiées car elles portent toutes les deux sur le premier appel de la fonction (par exemple, si une activité est appelée deux fois, les valeurs d'entrée (ou de sortie) du premier appel de l'activité ne correspondront pas forcément aux valeurs d'entrée (ou de sortie) de la fonction d'activité).

Un exemple de *behavior* pour l'appel de l'activité `pump_off` dans l'état `GearUp` est donné Listing B.4.

```
behavior Activity_GearUp :
assumes current_state==GearUp;
requires pump_off_counter==0;
assigns pump_state, pump_dir, pump_off_observer[pump_off_counter],
        pump_off_counter;
ensures \old(pump_state)==pump_off_observer[0].in.pump_state;
ensures \old(pump_dir)==pump_off_observer[0].in.pump_dir;
ensures pump_state==pump_off_observer[0].out.pump_state;
ensures pump_dir==pump_off_observer[0].out.pump_dir;
```

Listing B.4 – *behavior* pour l'appel de l'activité `pump_off` dans l'état `GearUp`

Il faut également vérifier une dernière propriété :

(e) si un état ne possède d'activité ou si cet état n'est pas défini dans la conception, aucune activité n'est appelée pour cet état par la fonction d'activité.

Cette propriété (e) peut être vérifiée très simplement par un *behavior* ACSL composé d'une clause *assumes* portant sur la valeur de l'état courant et d'une clause *assigns* spécifiée à l'aide du mot-clé *\nothing* pour garantir qu'aucune variable du programme n'est alors modifiée. Ce *behavior* appliqué à la fonction d'activité de l'exemple du train d'atterrissage est donné dans le Listing B.5.

```
behavior OtherStates :
assumes current_state!=GearDown && current_state!=LoweringGear
       && current_state!=StartLoweringGear && current_state!=GearUp
         && current_state!=RaisingGear && current_state!=StartRaisingGear
           && current_state!=WaitingForTakeoff;
assigns \nothing;
```

Listing B.5 – *behavior* pour l'appel de l'activité `pump_off` dans l'état `GearUp`

B.2 Conclusion

Cette méthode donne des résultats concluant sur l'exemple du train d'atterrissage. En ce qui concerne la génération automatique de ces annotations, la principale difficulté réside dans la récupération des variables modifiées par une activité. Il n'est plus question de parcourir uniquement la machine à états UML pour récupérer les annotations, il faut désormais parcourir les activités pour récupérer ces données. La principale problématique à résoudre est comment considérer qu'une variable globale est modifiée dans une activité ? (Est-ce une action particulière ? Est-ce un attribut d'une action en particulier ? Avons-nous un patron de modélisation indentifiable pour récupérer ce type de donnée ?). Des travaux supplémentaires seront nécessaires pour y répondre.

Annexe C

Questionnaire sur les méthodes formelles pour la vérification de programmes

Ce court questionnaire porte sur l'utilisation des méthodes formelles pour la vérification de programmes. La vérification formelle de programmes permet de prouver la conformité d'un programme par rapport à des propriétés ou de détecter des possibles erreurs à l'exécution. Citons à titre d'exemple : la méthode B, l'analyse statique de code (Frama-C, Astrée, Caveat, Why3...).

C.1 Connaissances

1. Connaissez-vous des méthodes formelles appliquées aux tâches de vérification de programmes ? **OUI/NON** (Si OUI, passage en SectionC.2 ; si NON, passage en Section C.3)

C.2 Expérience

1. Avez-vous déjà utilisé des méthodes formelles pour la vérification de programmes ? **OUI/NON**
2. D'après vous, quelle est la principale difficulté rencontrée lors de l'utilisation des méthodes formelles pour la vérification de programmes ? **Exprimer et formaliser les propriétés- Comprendre les résultats fournis – Il n'y a pas de difficulté – Sans opinion - autres (réponse libre)**
3. Trouvez-vous qu'il y ait un avantage à utiliser les méthodes formelles pour la vérification de programmes ? **PAS DU TOUT D'ACCORD 1 – 2 – 3 – 4 TOUT A FAIT D'ACCORD**

4. Quel investissement en formation pensez-vous être nécessaire pour l'utilisation des méthodes formelles ? **PAS DU TOUT D'INVESTISSEMENT 1 – 2 – 3 – 4 BEAUCOUP D'INVESTISSEMENT**

C.3 Attentes

1. Utiliseriez vous (ou utilisez vous déjà) des méthodes formelles pour la vérification de programmes afin de remplacer les techniques de vérification actuelles dans un processus de développement ? **OUI – NON –SANS OPINION**
2. Utiliseriez vous (ou utilisez vous déjà) des méthodes formelles pour la vérification de programmes afin d'ajouter une vérification supplémentaire dans un processus de développement pour améliorer la qualité du logiciel produit ? **OUI - NON – SANS OPINION**
3. Pour vous est-il envisageable qu'un utilisateur, dans le cadre d'une vérification de programmes, utilise les résultats donnés par un outil de preuve sans en connaitre les fondements théoriques ? (l'outil donne un résultat de vérification en masquant les détails de l'analyse formelle) **PAS DU TOUT D'ACCORD 1 – 2 – 3 – 4 TOUT A FAIT D'ACCORD**
4. Est-ce que vous pensez que dans le cadre d'une vérification de programmes réalisée à l'aide des méthodes formelles, il est indispensable que l'utilisateur soit un expert de ces méthodes pour légitimer les résultats de la vérification ? **PAS DU TOUT D'ACCORD 1 – 2 – 3 – 4 TOUT A FAIT D'ACCORD**
5. Si un outil basé sur les méthodes formelles permettant de vérifier automatiquement un programme par rapport à sa spécification vous était proposé, l'utiliseriez-vous ? **PAS DU TOUT 1 – 2 – 3 – 4 ABSOLUMENT**
6. Remarques ou suggestions sur l'utilisation des méthodes formelles dans le cadre de la vérification de programmes ? **REPONSE LIBRE (non obligatoire)**

C.4 Informations statistiques

1. Age **REPONSE LIBRE**
2. Profession **REPONSE LIBRE**
3. Niveau de formation **LICENSE – MASTER/INGENIEUR – DOCTORAT –AUTRE**

Bibliographie

[1] IEEE Standard Glossary of Software Engineering Terminology. Technical report, 1990.

[2] L. T. W. Agner, I. W. Soares, P. C. Stadzisz, and J. M. Simão. A brazilian survey on UML and model-driven practices for embedded software development. *Journal of Systems and Software*, 86(4) :997–1005, 2013.

[3] R. Alur and D. Dill. Automata for modeling real-time systems. In M. Paterson, editor, *Automata, Languages and Programming*, volume 443 of *Lecture Notes in Computer Science*, pages 322–335. Springer Berlin Heidelberg, 1990.

[4] R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126 :183–235, 1994.

[5] K. Anastasakis, B. Bordbar, G. Georg, and I. Ray. UML2Alloy : A Challenging Model Transformation. In *ACM/IEEE 10th International Conference on Model Driven Engineering Languages and Systems*, volume 4735 of *LNCS*, 2007.

[6] D. B. Aredo. Semantics of UML statecharts in PVS. In *Proc. of the 12th Nordic Workshop on Programming Theory*, NWPT'00, Denmark, 2000.

[7] A. Arnold. Systèmes de transitions finis et sémantique des processus communicants. *TSI. Technique et science informatiques*, 9(3) :193–216, 1990.

[8] C. Atkinson and T. Kuhne. Model-driven development : a metamodeling foundation. *Software, IEEE*, 20(5) :36–41, 2003.

[9] P. Baker, S. Loh, and F. Weil. Model-driven engineering in a large industrial context - Motorola case study. In L. Briand and C. Williams, editors, *Model Driven Engineering Languages and Systems*, volume 3713 of *Lecture Notes in Computer Science*, pages 476–491. Springer Berlin Heidelberg, 2005.

[10] E. Barbier, N. Cariou, and F. Belloir. Contrats de transformation pour la validation de raffinement de modèles. *IDM 2009 Actes des 5emes journées sur l'Ingénierie Dirigée par les Modèles*, pages 1–16, 2009.

[11] P. Baudin, L. Correnson, and Z. Dargaye. *WP Plug-in Manual version 0.7.* CEA LIST, 2013.

[12] P. Baudin, P. Cuoq, J. Filliâtre, C. Marché, B. Monate, Y. Moy, and V. Prevosto. *ACSL Version 1.7*, 2013.

[13] P. Baudin, A. Pacalet, J. Raguideau, D. Schoen, and N. Williams. Caveat : A tool for software validation. In *Proceedings of the 2002 International Conference on Dependable Systems and Networks*, DSN'02, Washington, DC, USA, 2002. IEEE Computer Society.

[14] G. Behrmann, A. David, and K. Larsen. A tutorial on UPPAAL. In M. Bernardo and F. Corradini, editors, *Formal Methods for the Design of Real-Time Systems*, volume 3185 of *LNCS*, pages 33–35. Springer Berlin / Heidelberg, 2004.

[15] G. Berry and G. Gonthier. The Esterel synchronous programming language : design, semantics, implementation. *Science of Computer Programming*, 19(2) :87 – 152, 1992.

[16] B. Berthomieu, P.-O. Ribet, and F. Vernadat. The tool TINA - construction of abstract state spaces for petri nets and time petri nets. *International Journal of Production Research*, 42(14) :2741–2756, 2004.

[17] J. Bézivin. In search of a basic principle for model driven engineering. *Novatica Journal*, pages 21–24, 2004.

[18] J. Bézivin, M. Blay, M. Bouzhegoub, J. Estublier, J.-M. Favre, S. Gérard, and J. M. Jézéquel. Rapport de synthèse de l'AS CNRS sur le MDA (model driven architecture). Technical report, CNRS, Novembre 2004.

[19] E. Börger, A. Cavarra, and E. Riccobene. Modeling the dynamics of UML state machines. In Y. Gurevich, P. Kutter, M. Odersky, and L. Thiele, editors, *Abstract State Machines - Theory and Applications*, volume 1912 of *Lecture Notes in Computer Science*, pages 223–241. Springer Berlin Heidelberg, 2000.

[20] J.-L. Boulanger. *Utilisations industrielles des techniques formelles : interprétation abstraite*. Hermès science publications-Lavoisier, 2011.

[21] J.-L. Boulanger. *Industrial Use of Formal Methods : Formal Verification*. John Wiley & Sons, 2012.

[22] J. R. Büchi. On a Decision Method in a Restricted Second Order Arithmetic. In Press, editor, *Proceedings of the International Congress on Logic, Methodology and Philosophy of Science*, pages 1–11, 1960.

[23] J. Bézivin and O. Gerbé. Towards a precise definition of the OMG/MDA framework. In *Automated Software Engineering, 2001. (ASE 2001). Proceedings. 16th Annual International Conference on*, pages 273–280, 2001.

[24] J. Cabot, R. Clarisó, and D. Riera. Verification of UML/OCL class diagrams using constraint programming. In *ICSTW '08 : Proceedings of the 2008 IEEE International Conference on Software Testing Verification and Validation Workshop*, 2008.

[25] C. Chamontin. Etude et amélioration des retours d'analyse statique de code C. Rapport de stage ONERA, 2013.

[26] B. Combemale. *Approche de métamodélisation pour la simulation et la vérification de modèle–Application à l'ingénierie des procédés*. PhD thesis, Institut National Polytechnique de Toulouse-INPT, 2008.

[27] P. Cousot. *Méthodes itératives de construction et d'approximation de points fixes d'opérateurs monotones sur un treillis, analyse sémantique des programmes.* PhD thesis, Université scientifique et médicale de Grenoble, 1978.

[28] P. Cousot. La vérification des programmes par interprétation abstraite. Séminaire de la Chaire d'Innovation technologique – Liliane Bettencourt, Collège de France, 22 2008.

[29] P. Cousot and R. Cousot. Abstract interpretation : a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, 1977. ACM Press, New York, NY.

[30] P. Cousot and R. Cousot. *Basic Concepts of Abstract Interpretation*, pages 359–366. Kluwer Academic Publishers, 2004.

[31] M. L. Crane and J. Dingel. On the Semantics of UML State Machines : Categorization and Comparison. Technical Report 2005-501, School of Computing, Queen's University, Kingston, Ontario, Canada, 2005.

[32] K. Czarnecki and S. Helsen. Classification of model transformation approaches. In *Proceedings of the 2nd OOPSLA Workshop on Generative Techniques in the Context of the Model Driven Architecture*, volume 45, 2003.

[33] M. de Roquemaurel, T. Polacsek, J.-F. Rolland, J.-P. Bodeveix, and M. Filali. Assistance à la conception de modèles à l'aide de contraintes. In *10es Journées Francophones Internationales sur les Approches Formelles dans l'Assistance au Développement de Logiciels (AFADL'10)*, 2010.

[34] R. Delmas, D. Doose, A. Fernandes Pires, and T. Polacsek. Supporting model based design. In L. Bellatreche and F. Mota Pinto, editors, *Model and Data Engineering*, volume 6918 of *Lecture Notes in Computer Science*, pages 237–248. Springer Berlin Heidelberg, 2011.

[35] R. Delmas, A. Fernandes Pires, and T. Polacsek. A Verification & Validation process for Model Driven Engineering. In *Progress in Flight dynamics, guidance, navigation, control, fault detection, and avionics*, EUCASS.

[36] R. Delmas, T. Polacsek, D. Doose, and A. Fernandes Pires. IDM : vers une aide à la conception. In *Inforsid*, pages 147–162, Lille, France, Avril 2011.

[37] P. Dhaussy, J. Roger, and F. Boniol. Reducing state explosion with context modeling for model-checking. In *IEEE 13th International Symposium on High-Assurance Systems Engineering (HASE)*, pages 130–137, 2011.

[38] A. Dieumegard and M. Pantel. Vérification d'un générateur de code par génération d'annotations. In *Conférence en Ingénierie du Logiciel*, France, 2012.

[39] E. W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8) :453–457, Aug. 1975.

[40] E. Domínguez, B. Pérez, A. L. Rubio, and M. A. Zapata. A systematic review of code generation proposals from state machine specifications. *Information and Software Technology*, 54(10) :1045 – 1066, 2012.

[41] S. Duprat, P. Gaufillet, V. Moya Lamiel, and F. Passarello. Formal verification of SAM state machine implementation. In *ERTS*, France, 2010.

[42] M. H. Essoussi, A. Fernandes Pires, and S. Duprat. Preuve formelle de code à partir de diagrammes d'activités UML. In N. Plouzeau and P. Poizat, editors, *Actes de la 2ème Conférence en IngénieriE du Logiciel (CIEL)*, 2013.

[43] P. Farail, P. Gaufillet, A. Canals, C. Le Camus, D. Sciamma, P. Michel, X. Crégut, and M. Pantel. the TOPCASED project : a toolkit in open source for critical aeronautic systems design. In *Embedded Real Time Software (ERTS)*, pages 54–59, France, 2006.

[44] R. Faudou, T. Faure, S. Gabel, and C. Mertz. Topcased requirement : a model-driven, open-source and generic solution to manage requirement traceability. In *Embedded Real Time Software (ERTS)*, France, 2010.

[45] J.-M. Favre. Foundations of meta-pyramids : Languages vs. metamodels - episode II : Story of Thotus the baboon. In *Language Engineering for Model-Driven Software Development, number 04101 in Dagstuhl Seminar Proceedings. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl*, 2004.

[46] J.-M. Favre, J. Estublier, and M. Blay-Fornarino. *L'ingénierie dirigée par les modèles. Au-delà du MDA (Traité IC2, série Informatique et Systèmes d'Information)*. Traité IC2, série Informatique et Systèmes d'Information. Lavoisier, 2006.

[47] P. H. Feiler and D. P. Gluch. *Model-Based Engineering with AADL : An Introduction to the SAE Architecture Analysis & Design Language*. Addison-Wesley Professional, 1st edition, 2012.

[48] A. Ferlin and V. Wiels. Combination of static and dynamic analyses for the certification of avionics software. In *23rd IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, pages 331–336, 2012.

[49] A. Fernandes Pires. Apports des méthodes formelles pour les cycles de développement logiciel embarqué basés sur le modèle en V. In *Conférence en IngénieriE du Logiciel (CIEL)*, France, 2012.

[50] A. Fernandes Pires, S. Duprat, and C. Besseyre. Approche UML/SysML pour la spécification logicielle de systèmes embarqués aéronautiques : Travaux et retours d'expérience. *TSI. Technique et science informatiques*, 31(7) :897–916, 2012.

[51] A. Fernandes Pires, S. Duprat, T. Faure, C. Besseyre, J. Beringuier, and J.-F. Rolland. Use of modelling methods and tools in an industrial embedded system project : works and feedback. In *ERTS*, France, 2012.

[52] A. Fernandes Pires, T. Polacsek, and S. Duprat. Formal software verification at model and at source code levels. In A. Abelló, L. Bellatreche, and B. Benatallah,

editors, *MEDI*, volume 7602 of *Lecture Notes in Computer Science*, pages 162–169. Springer, 2012.

[53] A. Fernandes Pires, T. Polacsek, and S. Duprat. An Eclipse plug-in to link modeling and code proof, AGrUM : ACSL generator from UML model. In B. Carré, H. Sahroui, and H. Störrle, editors, *Proceedings of the Joint Track "Tools, Demos, and Posters" of ECOOP, ECSA, and ECMFA 2013*, 2014.

[54] A. Fernandes Pires, T. Polacsek, V. Wiels, and S. Duprat. Behavioural verification in embedded software, from model to source code. In A. Moreira, B. Schätz, J. Gray, A. Vallecillo, and P. Clarke, editors, *Model-Driven Engineering Languages and Systems*, volume 8107 of *Lecture Notes in Computer Science*, pages 320–335. Springer Berlin Heidelberg, 2013.

[55] A. Fernandes Pires, T. Polacsek, V. Wiels, and S. Duprat. Vérifier le comportement du code d'un système embarqué à partir de son modèle. In *Modélisation des systèmes réactifs - Actes de MSR 2013*, volume 47 of *Journal Européen des Systèmes Automatisés (JESA)*, pages 61–75. Lavoisier, 2013.

[56] A. Fernandes Pires, T. Polacsek, V. Wiels, and S. Duprat. Use of formal methods in embedded software development : stakes, constraints and proposal. In *ERTS*, France, 2014.

[57] J.-C. Filliâtre and A. Paskevich. Why3 — where programs meet provers. In M. Felleisen and P. Gardner, editors, *Proceedings of the 22nd European Symposium on Programming*, volume 7792 of *Lecture Notes in Computer Science*, pages 125–128. Springer, Mar. 2013.

[58] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for java. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, PLDI '02, pages 234–245, New York, NY, USA, 2002. ACM.

[59] R. W. Floyd. Assigning meanings to programs. *Mathematical aspects of computer science*, 19(19-32) :1, 1967.

[60] A. Forward and T. C. Lethbridge. Problems and opportunities for model-centric versus code-centric software development : a survey of software professionals. In *Proceedings of the 2008 international workshop on Models in software engineering*, MiSE '08, pages 27–32, New York, NY, USA, 2008. ACM.

[61] V. Fredriksen. Wide gap amongst developers' perception of the importance of uml tools, developereye study reveals. Express Press Release, 2005.

[62] P. Froment. L'architecture avionique de l'A380. *Réalités industrielles*, (NOV) :45–48, 2005.

[63] J. H. Gallier. *Logic for Computer Science : Foundations of Automatic Theorem Proving*, chapter 10, pages 448–476. Wiley, 1987.

[64] A. Gerber, M. Lawley, K. Raymond, J. Steel, and A. Wood. Transformation : The missing link of MDA. In A. Corradini, H. Ehrig, H.-J. Kreowski, and G. Rozenberg,

editors, *Graph Transformation*, volume 2505 of *Lecture Notes in Computer Science*, pages 90–105. Springer Berlin Heidelberg, 2002.

[65] M. Gogolla and F. P. Presicce. State diagrams in UML : A formal semantics using graph transformations - or diagrams are nice, but graphs are worth their price. Technical Report TUM-I9803, 1998, Technical University of Munich, 1998.

[66] M. Gomez. Embedded state machine implementation. *Embedded Systems Programming*, page 41, 2000.

[67] R. C. Gronback. *Eclipse Modeling Project : A Domain-Specific Language (DSL) Toolkit*. Addison-Wesley Professional, 1 edition, 2009.

[68] S. Gérard, H. Espinoza, F. Terrier, and B. Selic. 6 modeling languages for real-time and embedded systems. In H. Giese, G. Karsai, E. Lee, B. Rumpe, and B. Schätz, editors, *New Technologies of Distributed Systems (NOTERE)*, volume 6100 of *Lecture Notes in Computer Science*, pages 129–154. Springer Berlin / Heidelberg, 2011.

[69] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data flow programming language LUSTRE. *Proceedings of the IEEE*, 79(9) :1305–1320, Sep 1991.

[70] D. Harel. Statecharts : A visual formalism for complex systems. In *Science of Computer Programming*, volume 8, pages 321–274, 1987.

[71] D. Harel and H. Kugler. The rhapsody semantics of statecharts (or, on the executable core of the uml). In H. Ehrig, W. Damm, J. Desel, M. Große-Rhode, W. Reif, E. Schnieder, and E. Westkämper, editors, *Integration of Software Specification Techniques for Applications in Engineering*, volume 3147 of *Lecture Notes in Computer Science*, pages 325–354. Springer Berlin Heidelberg, 2004.

[72] D. Harel and M. Politi. *Modeling Reactive Systems with Statecharts : The Statemate Approach*. McGraw-Hill, New York, NY, USA, 1998.

[73] A. Hartman. Model based test generation tools. *Agedis Consortium*, 2002.

[74] K. Havelund and T. Pressburger. Model checking java programs using java pathfinder. *International Journal on Software Tools for Technology Transfer*, 2(4) :366–381, 2000.

[75] P. Herrmann and J. Signoles. *Frama-C's annotation generator plug-in for Frama-C Fluorine*, 2013.

[76] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10) :576–580, 1969.

[77] J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. Pearson International Edition, 3 edition, 2006.

[78] E. Hubbers and M. Oostdijk. Generating JML specifications from UML state diagrams. In *In Forum on Specification & Design Languages FDL'03*, pages 263–273, 2003.

[79] D. Huffman. The synthesis of sequential switching circuits. *Journal of the Franklin Institute*, 257(3) :161–190, 1954.

[80] R. Jobredeaux, T. Wang, and E. Feron. Autocoding control software with proofs I : Annotation translation. In *Digital Avionics Systems Conference (DASC), 2011 IEEE/AIAA 30th*, pages 7C1–1–7C1–13, Oct.

[81] N. J. Juzgado, A. M. Moreno, and W. Strigel. Guest editors' introduction : Software testing practices in industry. *IEEE Software*, 23(4) :19–21, 2006.

[82] S. C. Kleene. Representation of Events in Nerve Nets and Finite Automata. In C. Shannon and J. McCarthy, editors, *Automata Studies, Annals of Math. Studies 34*. New Jersey, 1956.

[83] N. Kosmatov and J. Signoles. A lesson on runtime assertion checking with Frama-C. In A. Legay and S. Bensalem, editors, *Runtime Verification*, volume 8174 of *Lecture Notes in Computer Science*, pages 386–399. Springer Berlin Heidelberg, 2013.

[84] Y. Ledru, L. du Bousquet, P. Bontron, O. Maury, C. Oriat, and M.-L. Potet. Test purposes : adapting the notion of specification to testing. In *Proceedings of the 16th Annual International Conference on Automated Software Engineering (ASE)*, pages 127–134, Nov 2001.

[85] X. Leroy. A formally verified compiler back-end. *Journal of Automated Reasoning*, 43(4) :363–446, 2009.

[86] X. Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7) :107–115, 2009.

[87] J. Lilius and I. P. Paltor. Formalising UML state machines for model checking. In R. France and B. Rumpe, editors, *UML'99 - The Unified Modeling Language*, volume 1723 of *Lecture Notes in Computer Science*, pages 430–444. Springer Berlin Heidelberg, 1999.

[88] G. H. Mealy. A Method for Synthesizing Sequential Circuits. *Bell System Technical Journal*, 34(5) :1045–1079, 1955.

[89] B. Meyer. On formalism in specifications. *IEEE Softw.*, 1985.

[90] P. Mohagheghi and V. Dehlen. Where is the proof ? - a review of experiences from applying mde in industry. In *Proceedings of the 4th European conference on Model Driven Architecture : Foundations and Applications*, ECMDA-FA '08, pages 432–443, Berlin, Heidelberg, 2008. Springer-Verlag.

[91] E. F. Moore. Gedanken Experiments on Sequential Machines. In *Automata Studies*, pages 129–153. Princeton U., 1956.

[92] Y. Moy, E. Ledinot, H. Delseny, V. Wiels, and B. Monate. Testing or formal verification : Do-178c alternatives and industrial experience. *IEEE Software*, 30(3) :50–57, 2013.

[93] G. J. Myers. *The Art of Software Testing, Second Edition.* Wiley, 2 edition, 2004.

[94] A. Olivé. State transition diagrams. In *Conceptual Modeling of Information Systems*, pages 299–323. Springer, 2007.

[95] OMG. OMG Meta Object Facility (MOF) specification v1.4, 2002.

[96] OMG. *MDA Guide Version 1.0. 1*, 2003.

[97] OMG. *Business Process Model and Notation (BPMN) 2.0*, 2011.

[98] OMG. *Unified Modeling Language (UML) Infrastructure Specification 2.4.1*, 2011.

[99] OMG. *Object Constraint Language (OCL) 2.3.1*, 2012.

[100] OMG. *Systems Modeling Language (SysML) Specification 1.3*, 2012.

[101] D. Pariente and E. Ledinot. Formal verification of industrial C code using Frama-C : a case study. In *Formal Verification of Object-Oriented Software*, 2010.

[102] G. Pedroza, L. Apvrille, and D. Knorreck. AVATAR : A SysML environment for the formal verification of safety and security properties. In *2011 11th Annual International Conference on New Technologies of Distributed Systems (NOTERE)*, pages 1–10, 2011.

[103] D. Perrin. Les débuts de la théorie des automates. *TSI. Technique et science informatiques*, 14(4) :409–433, 1995.

[104] J. L. Peterson. *Petri Net Theory and the Modeling of Systems*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1981.

[105] M. Petre. UML in practice. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 722–731, Piscataway, NJ, USA, 2013. IEEE Press.

[106] R. Pettit. Lessons learned applying uml in embedded software systems designs. In *Proceedings of the Second IEEE Workshop on Software Technologies for Future Embedded and Ubiquitous Systems*, pages 75–79, 2004.

[107] D. Potier. Briques génériques du logiciel embarqué. Technical report, Ministère de l'industrie (France), 2010.

[108] J. Pouly and S. Jouanneau. Model-based specification of the flight software of an autonomous satellite. In *ERTS*, 2012.

[109] M. O. Rabin and D. Scott. Finite automata and their decision problems. *IBM Journal of Research and Development*, 3(2) :114–125, 1959.

[110] C. Ramchandani. *Analysis of Asynchronous Concurrent Systems By Timed Petri Nets*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, 1974.

[111] G. Reggio, E. Astesiano, C. Choppy, and H. Hussmann. Analysing UML Active Classes and Associated State Machines - A Lightweight Formal Approach. In T. Maibaum, editor, *Fundamental Approaches to Software Engineering*, volume 1783 of *FASE '00*, pages 127–146. Springer Berlin Heidelberg, UK, 2000.

[112] RTCA/EUROCAE. DO-178C/ED-12C : Software Considerations in Airborne Systems and Equipment Certification, 2011.

[113] RTCA/EUROCAE. DO-333/ED-216 : Formal Methods Supplement to DO-178C and DO-278A, 2011.

[114] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language reference manual*. Pearson Higher Education.

[115] P. Runeson. A Survey of Unit Testing Practices. *IEEE Software*, 23(4) :22–29, July 2006.

[116] M. Samek. *Practical UML Statecharts in C/C++, Second Edition : Event-Driven Programming for Embedded Systems*. Newnes, Newton, MA, USA, 2 edition, 2008.

[117] B. Selic. The pragmatics of model-driven development. *Software, IEEE*, 20(5) :19–25, 2003.

[118] C. E. Shannon. A mathematical theory of communication. *The Bell Systems Technical Journal*, 27 :379.423, 623.656, 1948.

[119] M. Soeken, R. Wille, M. Kuhlmann, M. Gogolla, and R. Drechsler. Verifying uml/ocl models using boolean satisability. In *Wolfgang Müller, editor, Proc. Design, Automation and Test in Europe (DATE'2010)*, 2010.

[120] R. Soley et al. Model driven architecture. White paper, OMG.

[121] J. Souyris, V. Wiels, D. Delmas, and H. Delseny. Formal verification of avionics software products. In A. Cavalcanti and D. Dams, editors, *FM 2009 : Formal Methods*, volume 5850 of *Lecture Notes in Computer Science*, pages 532–546. Springer Berlin Heidelberg, 2009.

[122] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks. *EMF : Eclipse Modeling Framework 2.0*. Addison-Wesley Professional, 2nd edition, 2009.

[123] H. Tardieu, A. Rochfeld, R. Colletti, and J. Lesourne. *La méthode MERISE : principes et outils*. Editions d'organisation, 1983.

[124] The Middleware Company on behalf of Compuware. Model driven development for J2EE utilizing a Model Driven Architecture (MDA) approach : Productivity analysis. Report, 2003.

[125] A. Toom, T. Naks, M. Pantel, M. Gandriau, and I. Wati. Gene-auto : an automatic code generator for a safe subset of simulink/stateflow and scicos. In *European Conference on Embedded Real-Time Software (ERTS'08)*, 2008.

[126] A. M. Turing. On computable numbers, with an application to the entscheidugsproblem. *Proceedings of the London Mathematical Society*, 42(2) :230–265, 1936-7.

[127] A. M. Turing. Computing Machinery and Intelligence. *Mind*, 59(236) :433–460, Oct. 1950.

[128] M. Utting and B. Legeard. *Practical model-based testing : a tools approach*. Morgan Kaufmann, 2010.

[129] D. Varró. A Formal Semantics of UML Statecharts by Model Transition Systems. In A. Corradini, H. Ehrig, H.-J. Kreowski, and G. Rozenberg, editors, *Proc. of the First International Conference on Graph Transformation*, ICGT '02, pages 378–392, UK, 2002. Springer-Verlag.

[130] T. Weigert and F. Weil. Practical experiences in using model-driven engineering to develop trustworthy computing systems. In *Sensor Networks, Ubiquitous, and Trustworthy Computing, 2006. IEEE International Conference on*, volume 1, 2006.

Résumé

Lors d'un développement logiciel, et plus particulièrement d'un développement d'applications embarquées avioniques, les activités de vérification représentent un coût élevé. Une des pistes prometteuses pour la réduction de ces coûts est l'utilisation de méthodes formelles. Ces méthodes s'appuient sur des fondements mathématiques et permettent d'effectuer des tâches de vérification à forte valeur ajoutée au cours du développement. Les méthodes formelles sont déjà utilisées dans l'industrie. Cependant, leur difficulté d'appréhension et la nécessité d'expertise pour leur mise en pratique sont un frein à leur utilisation massive.

Parallèlement au problème des coûts liés à la vérification logicielle, vient se greffer la complexification des logiciels et du contexte de développement. L'Ingénierie Dirigée par les Modèles (IDM) permet de faire face à ces difficultés en proposant des modèles, ainsi que des activités pour en tirer profit.

Le but des travaux présentés dans cette thèse est d'établir un lien entre les méthodes formelles et l'IDM afin de proposer à des utilisateurs non experts une approche de vérification formelle et automatique de programmes susceptible d'améliorer les processus de vérification actuels. Nous proposons de générer automatiquement sur le code source des annotations correspondant aux propriétés comportementales attendues du logiciel, et ce, à partir de son modèle de conception. Ces annotations peuvent ensuite être vérifiées par des outils de preuve déductive, afin de s'assurer que le comportement du code est conforme au modèle. Cette thèse CIFRE s'inscrit dans le cadre industriel d'Atos. Il est donc nécessaire de prendre en compte le contexte technique qui s'y rattache. Ainsi, nous utilisons le standard UML pour la modélisation, le langage C pour l'implémentation et l'outil Frama-C pour la preuve du code. Nous tenons également compte des contraintes du domaine du logiciel avionique dans lequel Atos est impliqué et notamment les contraintes liées à la certification.

Les contributions de cette thèse sont la définition d'un sous-ensemble des machines à états UML dédié à la conception comportementale de logiciel avionique et conforme aux pratiques industrielles existantes, la définition d'un patron d'implémentation C, la définition de patrons de génération des propriétés comportementales sur le code à partir du modèle et enfin l'implémentation de l'approche dans un prototype compatible avec l'environnement de travail des utilisateurs potentiels en lien avec Atos. L'approche proposée est finalement évaluée par rapport à l'objectif de départ, par rapport aux attentes de la communauté du génie logiciel et par rapport aux travaux connexes.

Abstract

During software development, and more specifically embedded avionics applications development, verification is very expensive. A promising lead to reduce its costs is the use of formal methods. Formal methods are mathematical techniques which allow performing rigorous and high-valued verification tasks during software development. They are already applied in industry. However, the high level of expertise required for their use is a major obstacle for their massive use.

In addition to the verification costs issue, today software and their development are subject to an increase in complexity. Model Driven Engineering (MDE) allows dealing with these difficulties by offering models, and tasks to capitalize on these models all along the development lifecycle.

The goal of this PhD thesis is to establish a link between formal methods and MDE in order to propose to non-expert users a formal and automatic software verification approach which helps to improve software verification processes. We propose to automatically generate annotations, corresponding to the expected behavioural properties of the software, from the design model to the source code. Then, these annotations can be verified using deductive proof tools in order to ensure that the behaviour of the code conforms to the design model. This PhD thesis takes place in the industrial context of Atos. So, it is necessary to take into account its technical specificities. We use UML for the design modeling, the C language for the software implementation and the Frama-C tool for the proof of this implementation. We also take into account the constraints of the avionics field in which Atos intervenes, and specifically the certification constraints.

The contributions of this PhD thesis are the definition of a subset of UML state machine dedicated to the behavioural design of embedded avionics software and in line with current industrial practices, the definition of a C implementation pattern, the definition of generation patterns for the behavioural properties from the design model to the source code and the implementation of the whole approach in a prototype in accordance with the working environment of the potential users associated with Atos. The proposed approach is then assessed with respect to the starting goal of the thesis, to the expectation of the software engineering community and to related work.