Apports des méthodes formelles pour les cycles de développement logiciel embarqué basés sur le modèle en V

Anthony Fernandes Pires
Atos/ONERA
Toulouse, France
anthony.fernandespires@atos.net

Abstract

Dans un cycle de développement logiciel, il n'est pas rare de voir plus de la moitié du temps de développement dédié à la vérification du logiciel et de sa conformité avec ses spécifications. Les méthodes formelles offrent de nouvelles possibilités pour la vérification tant au niveau de la spécification au travers par exemple de l'analyse des modèles par simulation ou du model-checking qui permet de détecter au plus tôt des erreurs possibles, qu'au niveau du code logiciel, où des techniques d'analyses, telle que l'interprétation abstraite, permettent de faciliter les étapes de vérification. Nous proposons ici d'améliorer les cycles de développement basés sur le modèle du cycle en V en y incorporant des phases de vérification à l'aide de méthodes formelles. Nous décrirons le contexte industriel de nos travaux, une description des différentes étapes de notre cycle et un état de l'art des techniques actuelles. Nous terminerons par des considérations sur les possibilités de couplage fort entre les différentes étapes de vérification de notre cycle.

Mots-clés : Vérification, méthodes formelles, cycle de développement, Ingénierie Dirigée par les Modèles

In a software development cycle, it is often more than half of the development time that is dedicated to the verification of its behavior and of its compliance with upstream specifications. Formal methods offer new possibilities for verification, in the specification phase with model analysis thanks to simulation or model-checking which allow users to detect errors in upstream phases, as well in the implementation phase, where analysis techniques, like abstract interpretation, make the verification tasks easier. In that context, we propose to take advantage of these methods to improve development cycle based on the V cycle. We will present the context of our work, a description of the different steps of our cycle and a state of the art of actual techniques. We will conclude with considerations on the possibilities of strong links between the different verification steps of our cycle.

Keywords : Verification, formal methods, development cycle, Model Based Engineering

1 Introduction

Au cours d'un cycle de développement classique de logiciels, les activités de vérification représentent une part importante du temps et des coûts. Au siècle dernier, [12] faisait le constat que plus de la moitié du temps de développement était consacré aux tests des programmes. Aujourd'hui, suite à l'accroissement de la taille des logiciels et de leur complexité, notamment dans le monde du logiciel embarqué, ces activités sont devenues un point crucial dans tout développement. Si [15] observe que 40 à 50% des coûts totaux de développement sont dédiés aux activités de vérification et validation, nos propres mesures au sein des projets que nous avons étudié montrent que ce coût peut parfois atteindre plus de 60% de la charge totale de travail.

Les méthodes formelles, parce qu'elles permettent de conduire des analyses exhaustives sur les systèmes afin de vérifier leurs comportements, offrent de nouvelles possibilités pour

ces activités. S'appuyant sur des bases mathématiques éprouvées, elles permettent de prouver l'exactitude de propriétés de conception. Ainsi, lors des phases amont d'un cycle de développement, les activités de simulation ou de model-checking sur des spécifications basées modèle permettent de détecter au plus tôt des erreurs possibles de fonctionnement et permettent d'obtenir une plus grande confiance dans les spécifications. Lors des phases aval, les techniques telles que l'analyse statique permettent de faciliter les activités d'analyse de code en détectant de possibles erreurs qui auraient lieu à l'exécution, participant ainsi à la robustesse du produit final.

Dans la droite ligne de la DO-333[1], nous proposons d'enrichir un cycle de développement logiciel embarqué basé sur le modèle du cycle en V en y introduisant des méthodes formelles. Nous voulons ici, non seulement introduire au plus tôt des phases de vérification, mais aussi introduire des techniques d'analyses formelles dans les phases de vérification du cycle en V en complément des tests classiques. Par ailleurs, de part le contexte industriel de ces travaux, il est nécessaire de prendre en compte les formalismes utilisés actuellement par les équipes projets ainsi que le savoir-faire des utilisateurs finaux. Ainsi, pour nous, il n'est pas possible de proposer un cycle basé sur des techniques introduisant une rupture avec les formalismes actuels et de demander à l'utilisateur de devenir un expert en méthodes formelles. Notre but est de faire cohabiter au mieux les formalismes utilisés, majoritairement issus de l'Ingénierie dirigée par les Modèles, et les techniques d'analyses formelles disponibles, tout en facilitant les activités des équipes projets. Notre contexte industriel, qui s'inscrit dans le domaine du logiciel embarqué, limite nos travaux aux standards UML/SysML pour les phases de spécification, standard reconnu et largement répandu, et aux différentes méthodes d'analyse statique proposées par le framework Frama-C[2] pour l'analyse de code.

Dans la suite de l'article, nous décrirons le cycle de développement que nous souhaitons mettre en place conformément à notre contexte puis nous présenterons un état de l'art non-exhaustif des techniques actuelles correspondantes aux besoins des différentes phases de notre cycle. Nous terminerons par une présentation des travaux à venir.

2 Notre cycle de développement logiciel

Le cycle en V est le modèle de cycle de vie classique employé pour le logiciel embarqué. Il est notamment adapté au cadre de la norme aéronautique DO-178C, qui définie les étapes de développement que doit respecter un logiciel avionique en vue de sa certification. Ce cycle se base sur une phase de spécification qui permet de formaliser les besoins client, suivie d'une phase de conception qui permet d'obtenir un premier prototype du système attendu, puis d'une phase d'implantation, phase centrale et de bas niveau, qui correspond à la réalisation du système lui-même. Suite à cette phase, il faut vérifier le système produit en passant tout d'abord par une phase de tests composée de tests unitaires sur le code ainsi que de tests d'intégration sur le programme global. Pour finir, une phase de validation est entreprise afin de s'assurer que le système correspond bien aux besoins attendus.

Le cycle que nous proposons Figure 1 est un enrichissement du cycle en V, et principalement de sa phase descendante en y incorporant des phases de vérification. Nous distinguerons la vérification qui permet de vérifier la conformité à la spécification et la validation qui permet de vérifier la conformité au besoin. Le cycle proposé débute par une phase de spécification sous forme de modèle. Le cycle se poursuit sur une phase de conception détaillée, qui s'apparente à

[1] Supplément de la norme aéronautique DO-178C pour le développement de logiciel embarqué, traitant plus particulièrement de l'utilisation des méthodes formelles dans le cycle de développement logiciel

[2] `http://frama-c.com/`

un raffinement des modèles et des propriétés exprimés dans les phases de spécification. Nous introduisons ensuite des vérifications dans les phases descendantes en introduisant une phase de vérification pour la conception, réalisée soit par simulation soit par model-checking, ou bien mixte en s'appuyant sur les techniques présentées en section 3.2. Cette phase de vérification en amont va permettre d'obtenir une plus grande confiance dans la spécification et vont également permettre de détecter de possibles problèmes qui pourraient se révéler plus coûteux s'ils n'apparaissaient qu'en phase aval du projet. Suite à ces phases, le cycle passe à une phase classique d'implantation du code source. Nous introduisons ensuite une phase de vérification du code source à partir d'analyses statiques comme nous l'introduisons en section 3.3. Pour cette phase, nous nous intéressons à la possibilité de passer des propriétés exprimées sur des modèles à des propriétés exprimées sur du code, mais ces travaux sont encore à venir. Le cycle continue ensuite par la phase classique de création de l'exécutable et les phases de tests et de validation.

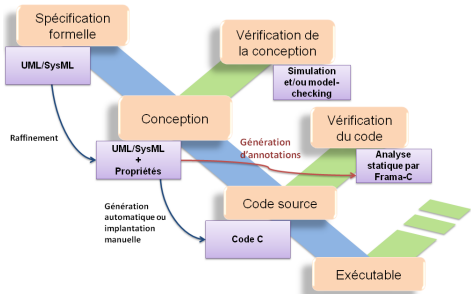

Figure 1: cycle proposé

3 Etat de l'art

3.1 Les langages pour spécifier

Il existe plusieurs moyens de spécifier un logiciel. L'usage du langage naturel est la méthode la plus classique. Il permet d'obtenir des spécifications qui peuvent être vérifiées par relecture mais qui sont souvent source d'ambiguïtés comme décrit dans [13]. Les langages de modélisation permettent d'obtenir des spécifications plus formelles mais demandent une formation. Ils offrent de nombreuses possibilités de vérification automatique et favorisent la communication sans ambiguïté entre partenaires. Une variété de standards existe de nos jours et ces langages peuvent être spécifiques à des domaines particuliers, comme la modélisation de logiciel, de système ou bien d'architectures de communications. Dans notre cas, nous travaillons avec des équipes qui sont familiarisées avec les standards UML/SysML. Ceux-ci sont génériques et permettent notamment de représenter les différents aspects d'un système ou d'un logiciel.

Certains travaux tendent à les spécialiser au domaine embarqué grâce à des mécanismes d'extension. Par exemple, le profil UML MARTE[3] [11] est un standard spécifié par l'OMG pour la modélisation des systèmes embarqués et phénomènes temps réel. Il spécifie des concepts pour caractériser des éléments UML permettant de représenter les notions de temps, de concurrence, de plateforme logicielle et matérielle, de ressources et de caractéristiques quantitatives sur un système, tel que le temps d'exécution. Il est organisé hiérarchiquement sous forme de packages

[3]Modeling and Analysis of Real-Time and Embedded Systems

regroupant chacun un aspect particulier du développement de logiciel embarqué temps réel. On distingue 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 le système.

Le profil SysML AVATAR [14] est un autre exemple de spécialisation au domaine embarqué qui offre des solutions pour la modélisation et la vérification formelle de système embarqué 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 de sûreté et de sécurité.

3.2 L'analyse formelle de modèle

L'intérêt de conduire des analyses dans des phases amont d'un cycle de développement est de détecter au plus tôt des problèmes de fonctionnement qui pourraient se révéler plus coûteux à corriger s'ils étaient détectés plus tardivement. Les modèles offrent une base formelle pour la représentation de système et l'expression de propriétés afin de conduire ces analyses et ils favorisent ainsi l'utilisation des méthodes formelles.

Le langage OCL, standard de l'OMG très répandu dans le monde de l'Ingénierie Dirigée par les Modèles, permet l'expression de propriétés sur des modèles UML. C'est un langage déclaratif qui ne permet pas d'effets de bord. Il propose un formalisme pour exprimer des contraintes, invariants, gardes, pré et post-conditions sur les éléments d'un modèle. Les vérifications de contraintes OCL sont possibles grâce à des OCL checkers. Néanmoins, comme le montre [5], il est possible de passer du monde du modèle vers d'autres techniques d'analyse formelle plus mathématiques pour une question de performance. Ainsi on peut transformer un modèle et des propriétés OCL vers des problèmes de satisfaction de contraintes et vérifier ces propriétés grâce aux techniques de résolution de ces problèmes. D'autres travaux tels que [17] et [7] proposent eux aussi la vérification de propriétés OCL, via une transformation vers des problèmes de résolution d'équations booléennes. L'utilisation de telles techniques offrent des apports en termes de performance pour la vérification de modèles de grandes tailles.

Les techniques de model-checking, introduites par [16] et [3], permettent de vérifier que le modèle d'un système respecte une spécification présentée sous forme d'un ensemble de propriétés. Dans le domaine embarqué, citons par exemple l'outil UPPAAL [1] qui propose, outre un model-checker, un environnement pour la modélisation, la simulation et la vérification de systèmes temps réel représentés sous forme d'automates temporisés. Le model-checker permet de vérifier des propriétés de sûreté de fonctionnement, d'accessibilité et de vivacité. En cas de propriétés non satisfaites, l'outil permet la génération de trace permettant de visualiser un contre-exemple par simulation. Des passerelles existent actuellement entre ce type d'outil et des formalismes type UML. Par exemple, l'outil TTool [6] donne la possibilité de vérifier des propriétés de sûreté de fonctionnement sur des modèles AVATAR grâce au model-checker UPPAAL.

3.3 L'analyse formelle de code

L'analyse statique de code permet d'obtenir des informations sur du code source sans l'exécuter, et notamment de détecter des erreurs possibles de fonctionnement ou de vérifier des propriétés.

Pour ce type d'analyse, nous nous intéressons principalement au framework Frama-C. Il se présente sous la forme d'un environnement modulaire open-source dédié à l'analyse statique de programmes C. Le framework s'appuie sur un langage d'annotation standardisé nommé ACSL [4], qui permet de définir des propriétés et des contrats sur les fonctions d'un programme, et il s'appuie également sur une palette de techniques d'analyses disponibles au travers de plugins et de solveurs.

La méthode d'analyse par valeur permet ainsi de s'assurer que le programme ne contient pas d'erreurs d'exécution. Elle se base sur le calcul des domaines de variations[5] des différentes variables d'un programme, et plus précisément sur du code séquentiel. Le plugin Value Analysis [2] de Frama-C implémente cette méthode. Il est capable de signaler des alarmes en cas de détection d'erreurs possibles d'exécution (par exemple, l'accès à une adresse invalide) ainsi que les domaines de variation des variables en différents points d'un programme.

La technique de vérification par calcul d'obligations de preuve dérivée de la notion de calcul de la plus petite précondition introduit par [8] permet de calculer pour un contrat fonctionnel constitué de pré et post-condition, la plus petite précondition pour que la post-condition soit valide[6]. Une fois ce calcul terminé, il faut prouver que la précondition initiale implique la plus petite précondition pour déduire que la post-condition est vraie. Ce type d'analyse peut être menée grâce aux plugins WP [4] ou Jessie [7].

D'autres techniques permettent de vérifier le fonctionnement d'un programme par rapport à un comportement spécifié sous forme d'automate. Dans ce sens, le plugin Aorai [18] permet d'annoter automatiquement un programme à partir d'un comportement d'automate pouvant être exprimé en logique temporelle linéaire (LTL). La vérification peut ensuite être menée par les techniques d'analyses décrites précédemment. Il garantit que si les annotations sont vérifiées, le comportement du programme est bien conforme au comportement de l'automate décrit en LTL. La vérification ne porte donc plus ici sur des vérifications unitaires de code, mais sur le séquencement global du programme. L'approche est prometteuse grâce à la proximité du format de spécification et du format de vérification.

4 Conclusion et perspectives

Les techniques d'analyse formelle sont nombreuses et peuvent intervenir à différentes étapes d'un cycle de développement en V. Le modèle de cycle que nous proposons est une première proposition et devra s'appuyer sur des techniques qui permettent, à chaque étape du développement, de détecter des erreurs qui pourraient s'avérer plus coûteuses si elles étaient détectées tardivement dans le cycle de développement. L'objectif est double: il faut pouvoir détecter ces erreurs tout en facilitant l'activité de l'utilisateur et permettre l'analyse de l'origine des erreurs.

Concernant les travaux futurs, nous souhaitons appliquer ce cycle sur un cas d'étude simple pour commencer. Le formalisme de départ, caractérisé par un sous-ensemble d'UML/SysML que nous avons défini dans [10], est spécifique à la spécification de logiciel embarqué. Il se base sur la représentation du comportement du logiciel à partir des diagrammes de machine à états et d'activité, communs à UML et SysML. Le choix des techniques et outils pour la phase de vérification de la conception reste encore à définir. Nous en avons donné un premier aperçu en accord avec notre contexte mais nous sommes conscients qu'il en existe d'autres à explorer. La phase de vérification de l'implantation représentera le point important de notre

[4] ANSI/ISO C Specification Language
[5] les différentes valeurs possibles prises au cours de l'exécution
[6] des variants et des invariants sont également nécessaires en cas de boucle ou de fonction récursive
[7] `http://krakatoa.lri.fr/jessie.html`

contribution. Nous nous intéresserons à la possibilité de s'appuyer sur des propriétés issues de la modélisation pour guider l'analyse de code. Des travaux existent sur le sujet, notamment [9] qui propose de déduire d'une spécification basée modèle des annotations pour réaliser l'analyse de l'implantation en s'appuyant sur l'outil Frama-C et le langage ACSL. Dans ce sens, nous nous intéresserons aux moyens de générer des annotations à partir d'un modèle UML/SysML pour vérifier des propriétés sur du code C à l'aide de techniques d'analyse statique et ainsi s'assurer de la conformité du code source avec les spécifications amont.

References

[1] G. Behrmann, A. David, and K. Larsen. A tutorial on uppaal. In *Formal Methods for the Design of Real-Time Systems*, volume 3185 of *Lecture Notes in Computer Science*, pages 33–35. Springer Berlin / Heidelberg, 2004.

[2] G. Canet, P. Cuoq, and B. Monate. A value analysis for c programs. In *SCAM '09*, pages 123–124, USA, 2009. IEEE Computer Society.

[3] Edmund M. Clarke and E.Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs, Workshop*, London, UK, 1982. Springer-Verlag.

[4] L. Correnson, Z. Dargaye, and A. Pacalet. *Frama-C WP Plug-in Manual*.

[5] 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 *AFADL'10*, 2010.

[6] P. De Saqui-Sannes and L. Apvrille. AVATAR/TTool : un environnement en mode libre pour SysML temps réel. *Génie logiciel*, 58(98):22–26, 2011.

[7] R. Delmas, T. Polacsek, D. Doose, and A. Fernandes Pires. Supporting model-based design. In *MEDI'11*, Portugal, Septembre 2011.

[8] E. W. Dijkstra. A constructive approach to program correctness. *BIT Numerical Mathematics*, 1968.

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

[10] 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.

[11] S. Gérard, H. Espinoza, F. Terrier, and B. Selic. 6 modeling languages for real-time and embedded systems. In *NOTERE*, volume 6100 of *Lecture Notes in Computer Science*, pages 129–154. Springer Berlin / Heidelberg, 2011.

[12] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 1969.

[13] B. Meyer. On formalism in specifications. *IEEE Softw.*, 2(1):6–26, January 1985.

[14] G. Pedroza, L. Apvrille, and D. Knorreck. Avatar : A sysml environment for the formal verification of safety and security properties. In *NOTERE*. 2011.

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

[16] J. Queille and J. Sifakis. Specification and verification of concurrent systems in cesar. In *International Symposium on Programming*, volume 137 of *Lecture Notes in Computer Science*, pages 337–351. Springer Berlin / Heidelberg, 1982.

[17] M. Soeken, R. Wille, M. Kuhlmann, M. Gogolla, and R. Drechsler. Verifying uml/ocl models using boolean satisfiability. In *DATE'2010*, 2010.

[18] N. Stouls and V. Prevosto. *Aoraï Plug-in Tutorial*.