Unification de la Vérification et de l'Exécution Embarquée de Modèles

Valentin Besnard

ERIS, ESEO-TECH, Angers, France, *valentin.besnard@eseo.fr*

Résumé

Pour lutter contre la complexité croissante des systèmes embarqués, les modèles de ces systèmes peuvent être vérifiés dès la phase de conception en utilisant diverses techniques de vérification et de validation (V&V). L'application de ces techniques requiert généralement deux transformations à partir du modèle de conception : une première pour obtenir le code exécutable du système et une seconde vers un modèle formel sur lequel pourront être appliqués les outils de V&V. Cependant, ces transformations créent des fossés sémantiques et nécessitent d'établir une relation d'équivalence entre le code exécutable et le modèle formel obtenu. Pour aborder ces problèmes, cet article présente une approche permettant d'exécuter et de vérifier des modèles avec une seule implémentation de la sémantique du langage de modélisation. L'exécution et la vérification de ces modèles est assurée par un interpréteur de modèles pilotable par des outils de V&V. Cette approche a été appliquée au langage UML avec l'interpréteur de modèles embarqué EMI et le model-checker OBP2.

Mots clés— Interprétation de modèles, Model-checking, Systèmes embarqués, UML

1 Introduction

Les systèmes embarqués deviennent de plus en plus complexes, ce qui rend leur conception plus difficile et les exposent davantage aux défaillances logicielles. Dans le contexte de l'ingénierie dirigée par les modèles, ces systèmes peuvent être représentés sous forme de modèles et analysés dès la phase de conception en appliquant des techniques de vérification et de validation (V&V).

Cependant, trois inconvénients majeurs subsistent généralement. D'abord, la transformation du modèle de conception en code exécutable crée un premier fossé sémantique entre ce modèle et le code généré. Ensuite, l'application des outils de V&V requiert généralement une seconde transformation vers un langage formel. Cette transformation crée un second fossé sémantique qui complexifie la compréhension des résultats de vérification. Enfin, une relation d'équivalence entre le modèle formel et le code exécutable doit être établie pour prouver que ce qui est exécuté est bien ce qui a été vérifié. Ces problèmes sont principalement dus aux transformations qui capturent la sémantique du langage de modélisation vers différents formalismes.

Pour lutter contre ces problèmes, cet article présente une approche permettant d'utiliser une seule implémentation de la sémantique du langage de modélisation pour l'exécution et la vérification de modèles. Cette sémantique est encodée dans un interpréteur de modèles pouvant être piloté par des outils de V&V. Cette approche permet ainsi d'appliquer la vérification directement sur le modèle interprété tout en réutilisant la même sémantique opérationnelle que celle utilisée pour l'exécution sur le système réel.

Notre approche a été mise en oeuvre avec le langage UML [18]. Un interpréteur de modèles embarqué, appelé EMI (Embedded Model Interpreter) [4, 5, 6], permet de simuler et vérifier des modèles UML avec le model-checker OBP2 (Observer-Based Prover 2) [20, 21] (`https://plug-obp.github.io/`), puis de les exécuter sur une cible embarquée STM32 discovery.

2 Approche

Pour mieux comprendre les problématiques et les enjeux abordés dans ce projet, notre approche est comparée avec l'approche classique de vérification et d'exécution embarquée de modèles. Cette comparaison permet de mettre en relief les spécificités et les atouts de l'approche proposée dans le contexte de l'exécution de modèles.

L'approche classique est illustrée sur la Figure 1. Le système à l'étude est modélisé sous la forme d'un *Modèle de Conception* dans le langage de modélisation choisi par l'équipe de développement. Le modèle doit donc se conformer au *Métamodèle* de ce langage. À partir du *Modèle de Conception*, l'approche classique utilise généralement deux transformations pour pouvoir exécuter et vérifier ce modèle. La première permet de transformer le *Modèle de Conception* en *Code* exécutable via de la génération de code automatique, semi-automatique, ou manuelle. Le *Code* produit peut ensuite être exécuté sur une *Cible Embarquée* et interagir avec l'environnement du système via les *Entrées/Sorties* de celle-ci. La seconde utilise des techniques de transformation de modèles pour produire un *Modèle d'Analyse* (ou modèle formel si exprimé dans un langage formel) à partir du *Modèle de Conception*. Ce *Modèle d'Analyse* peut ensuite être exploité par des *Outils de V&V Haut Niveau* afin de vérifier que le modèle satisfait aux exigences du système à un haut niveau d'abstraction.

Dans cette approche classique de développement embarqué, trois problèmes principaux subsistent. *(1)* La génération de code crée un premier fossé sémantique entre le modèle de conception et le code exécutable. Il devient ainsi plus difficile d'établir des liens entre les concepts du langage de modélisation et les fragments de code générés. Ce fossé sémantique a aussi un impact sur les outils de V&V puisque le modèle exprimé sous forme de code exécutable ne peut plus être analysé à un haut niveau d'abstraction et requiert l'utilisation d'*Outils de V&V Bas Niveau. (2)* Un second fossé sémantique est créé entre le modèle de conception et le modèle d'analyse. Il complexifie la compréhension des résultats de V&V en particulier pour des ingénieurs non-experts en méthodes formelles. Les activités de vérification peuvent aussi nécessiter de charger ce modèle d'analyse dans différents outils. Ce besoin induit un risque d'erreur supplémentaire si les différents processus de chargement ne sont pas exactement équivalents. *(3)* Une relation d'équivalence entre le modèle d'analyse et le code exécutable doit également est établie, prouvée, et maintenue afin d'assurer que les propriétés vérifiées en phase de V&V le soient aussi lors de l'exécution.

Ces trois problèmes sont principalement dus à l'utilisation de multiple implémentations de la sémantique du langage de modélisation. Ces multiples définitions proviennent

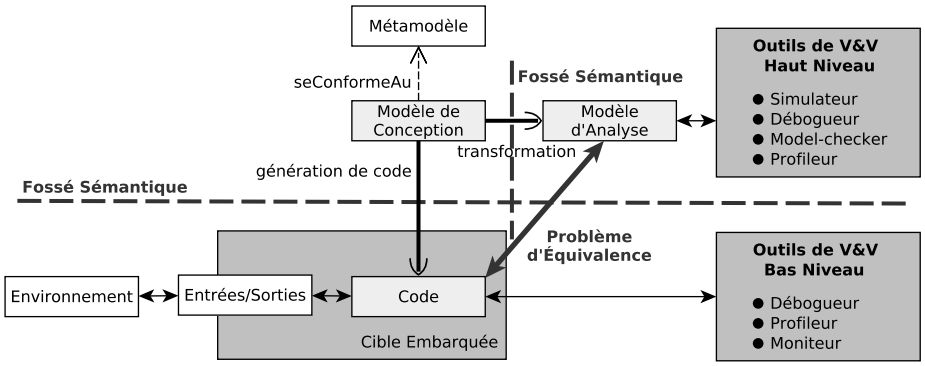


FIGURE 1 – Approche classique

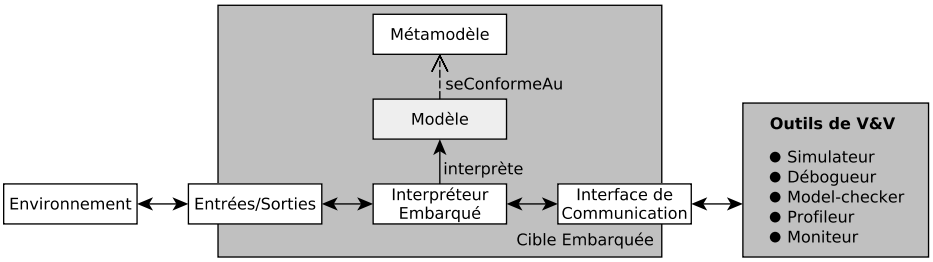


FIGURE 2 – Approche proposée

de l'utilisation de transformations qui capturent la sémantique du langage de modélisation en termes de la sémantique cible.

Pour traiter ces problèmes, la Figure 2 présente une nouvelle approche basée sur l'utilisation d'un interpréteur de modèles pilotable à distance par des outils de V&V. Dans cette approche, le *Modèle de Conception* et son *Métamodèle* sont directement chargés sur la *Cible Embarquée* afin d'éviter les deux transformations mentionnées précédemment. Le modèle est exécuté avec un *Interpréteur Embarqué* qui peut interagir avec l'environnement via les *Entrées/Sorties* de la cible embarquée comme dans l'approche classique. La spécificité de cet *Interpréteur Embarqué* est qu'une seule implémentation de la sémantique du langage est utilisée à la fois pour l'exécution et la vérification des modèles. Cette unique implémentation correspond à la sémantique opérationnelle de l'*Interpréteur Embarqué*. Pour la vérification, les *Outils de V&V* peuvent se connecter à l'*Interpréteur Embarqué* via une *Interface de Communication*. Les activités de V&V sont ainsi directement appliquées sur le modèle exécutable chargé par l'interpréteur tout en réutilisant la même implémentation de la sémantique du langage que celle utilisée pour l'exécution du système. Cet interpréteur de modèles doit pouvoir être déployé sur une cible embarquée mais doit aussi pouvoir être exécuté sur un ordinateur pour la phase de V&V afin d'obtenir des performances de vérification acceptables.

L'utilisation d'une seule implémentation de la sémantique du langage possède plusieurs avantages. Elle permet d'éviter les deux fossés sémantiques identifiés sur l'approche classique. Les résultats de vérification sont ainsi directement exprimés en termes du langage de modélisation ce qui permet de faciliter leur compréhension. Cette approche apporte

aussi une solution au problème d'équivalence et permet d'assurer que ce qui est exécuté est bien ce qui a été vérifié. Même en cas de bogue dans la sémantique opérationnelle de l'interpréteur, si le comportement du modèle satisfait aux exigences système alors celles-ci seront aussi satisfaites lors de l'exécution sur le système embarqué réel. En effet, le déploiement sur la cible embarquée utilise le même modèle et le même interpréteur que ceux utilisés pendant la phase de vérification. Cette approche tend donc à améliorer la qualité de développement et de vérification des systèmes embarqués dans le contexte de l'ingénierie dirigée par les modèles.

3 Application au Langage UML

L'approche proposée a été appliquée au langage UML et un interpréteur de modèles implémentant la sémantique de ce langage a été développé. Cet interpréteur de modèles, appelé EMI, est dédié à l'exécution et à la vérification de systèmes embarqués spécifiés sous la forme de modèles UML.

Bien qu'UML soit un langage semi-formel, l'approche reste valide car c'est la sémantique opérationnelle de l'interpréteur qui sert de référence. Par exemple, pour les points de variation sémantique d'UML, les choix d'implémentation faits dans l'interpréteur permettent de déterminer le comportement du système. Ces choix ont certes un impact sur la sémantique du langage mais cette même définition de la sémantique sera ensuite employée pour la vérification et l'exécution du système, préservant ainsi la relation d'équivalence entre le modèle d'analyse et le code exécutable.

Avant de pouvoir vérifier un modèle UML, plusieurs étapes sont nécessaires afin de pouvoir exécuter ce modèle sur l'interpréteur EMI. La première étape consiste à modéliser le système à concevoir en UML. Le modèle doit se conformer au sous-ensemble d'UML supporté par EMI. Ce sous-ensemble peut se représenter avec les diagrammes de classes, de structure composite, et d'états-transitions. Il permet de représenter à la fois la partie structurelle et la partie comportementale du modèle tout en exploitant partiellement les aspects orientés objet d'UML (e.g., héritage simple). Le modèle utilise aussi un langage d'action pour décrire les gardes et les effets des machines à états. Ce langage d'action est le langage C enrichi avec des macros C pour accéder aux éléments du modèle (e.g., les objets du modèle et leurs attributs). Une fois le modèle UML du système élaboré, il est sérialisé en langage C, le langage natif de l'interpréteur EMI. La sérialisation permet d'associer à chaque élément du modèle un initialiseur de structure C. Afin d'optimiser au mieux l'espace mémoire et les performances d'exécution, seuls les éléments du sous-ensemble considéré et nécessaires pour l'exécution du modèle sont pris en compte lors de la sérialisation. Contrairement à la génération de code, cette opération ne génère que des données mais pas de fonctions (sauf pour les expressions du langage d'action). Elle ne capture donc pas la sémantique du langage UML. Le modèle sérialisé en C et le code de l'interpréteur sont ensuite compilés et linkés avec un compilateur C pour produire le binaire exécutable de EMI. Cette opération peut être vue comme un chargement du modèle à la compilation. L'exécution de ce binaire exécutable permet d'interpréter le modèle UML chargé dans EMI. Cet interpréteur de modèles peut être exécuté sur un ordinateur équipé avec un OS Linux, ou en bare-metal (i.e., sans OS) sur une carte embarquée STM32 discovery.

Cet interpréteur de modèles UML peut ensuite être utilisé pour vérifier et analyser le comportement du modèle UML. Différentes activités de V&V peuvent être mises en oeuvre en connectant le model-checker OBP2 à l'interface de communication de EMI. Cette interface de communication permet de récupérer la configuration courante de l'interpréteur (i.e., la partie dynamique du modèle), de remettre l'interpréteur dans une confi-

guration donnée, de calculer l'ensemble des transitions tirables des machines à états du système, de tirer des transitions, et d'évaluer des prédicats. Cette interface permet aux outils de V&V de piloter l'interpréteur de modèles pour la vérification.

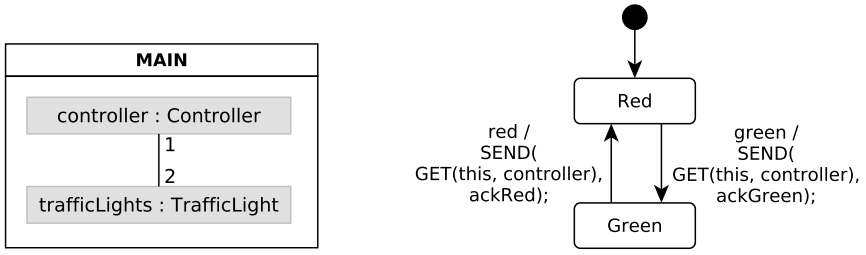
Le model-checker OBP2 permet d'appliquer plusieurs activités de V&V sur les modèles UML interprétés par EMI :
Simulation : OBP2 offre une interface de simulation permettant de visualiser et d'explorer différentes traces d'exécution. L'interface utilisateur de ce simulateur permet notamment de visualiser la configuration courante de l'interpréteur, de visualiser les transitions tirables, de tirer des transitions, et de visualiser les traces d'exécution déjà explorées.
Model-checking de propriétés LTL : Le model-checker OBP2 permet aussi de vérifier des propriétés LTL. Pour y parvenir, OBP2 explore l'espace d'états du système et utilise des techniques classiques de model-checking basées sur la composition d'automates de Büchi. Si une propriété est violée, le model-checker retourne un contre-exemple qui peut être visualisé sous la forme d'une trace dans l'interface de simulation.
Model-checking et monitoring avec des automates observateurs : Notre approche permet également d'exprimer des propriétés de sûreté en UML sous la forme d'automates observateurs. Ces automates sont composés de façon synchrone avec le système par l'interpréteur EMI et le model-checker n'a plus qu'à vérifier si les états de rejets de ces automates sont atteignables. Ces mêmes automates observateurs peuvent également être déployés avec EMI sur une cible embarquée pour faire du monitoring. Surveiller le comportement du système dans son environnement réel permet notamment de détecter des composants matériels défectueux, de réagir en cas de défaillances, et de faciliter l'analyse post-mortem.

La mise en oeuvre de ces activités de V&V avec EMI a ainsi permis de démontrer l'applicabilité de cette approche pour l'exécution et la vérification de modèles UML.

4 Exemple

Pour illustrer notre approche, cette section présente sa mise en oeuvre sur un exemple très simple. Il s'agit d'un modèle de gestion de deux feux de signalisation bicolores (vert et rouge) qui a pour objectif de garantir la sécurité des usagers et la fluidité de la circulation. Ce système a été modélisé en UML comme le montre le diagramme de composite structure et la machine à états de la classe *TrafficLight* en Figure 3. Ce modèle a ainsi pu être chargé dans EMI afin de vérifier les deux propriétés suivantes :

1. Le premier feu est au vert infiniment souvent.
2. Les deux feux ne sont jamais verts tous les deux simultanément.



(a) Diagramme de composite structure (b) Machine à états de *TrafficLight*

FIGURE 3 – Diagrammes d'un modèle de gestion de feux de signalisation.

Ces propriétés ont été traduites en LTL afin d'être vérifiées avec le model-checker OBP2 :

1. `"[] <> firstIsGreen"`
2. `"[] !(firstIsGreen and secondIsGreen)"`

Les proposition atomiques `firstIsGreen` et `secondIsGreen`, exprimées à l'aide du langage d'action C, permettent de savoir si respectivement le premier feu et le second feu sont verts à un instant donné. Ces deux propriétés ont été vérifiées avec succès sur le modèle UML qui a été conçu. Notre outil a également été validé sur des modèles plus complexes comme un contrôleur de passage à niveau [6].

5 État de l'Art

Ce projet se focalise sur l'exécution et la vérification de modèles UML dans le contexte des systèmes embarqués. D'autres outils permettent également d'exécuter et de vérifier des modèles UML comme le présente l'étude en [10].

Rational Software Architect [14] et Rhapsody [11] sont des outils de modélisation permettant de modéliser des modèles UML dans un environnement graphique, et de générer du code pour simuler et déboguer ces modèles. Les interpréteurs Moka [19] et Moliz [16] permettent d'interpréter des modèles conformes au standard fUML [17]. Ils peuvent être intégrés à l'outil de modélisation Papyrus [13] et être utilisés pour simuler, déboguer et tester des modèles fUML. GUML [8] et UniComp [9] sont des compilateurs de modèles permettant de compiler directement des modèles UML en code exécutable performant sans avoir besoin de passer par un formalisme intermédiaire. GEMOC Studio [7] est un framework permettant de concevoir un environnement de modélisation générique avec différents moteurs d'exécution et des outils de V&V (e.g., débogueur omniscient, animateur graphique). En comparaison avec notre interpréteur EMI, tous ces outils ne permettent pas d'appliquer des techniques formelles, comme le model-checking, sur des modèles de conception en utilisant leurs propres implémentations de la sémantique du langage.

D'autres outils permettent également d'analyser l'exécution des systèmes embarqués à un haut niveau d'abstraction comme les débogueurs [2] et [12]. Ces outils permettent de résoudre le problème du fossé sémantique entre le modèle de conception et le code exécutable mais ne permettent pas d'appliquer des méthodes formelles sur ces modèles en assurant que ce qui est vérifié est bien ce qui sera exécuté. Les compilateurs certifiés (e.g., CompCert [15]) permettent également de résoudre ce fossé sémantique en prouvant formellement que le code exécutable généré se comporte exactement comme le programme source.

Des approches alternatives permettent également de garantir la sûreté de fonctionnement du code exécutable. Event-B [1] est une méthode formelle permettant la modélisation et l'analyse de systèmes. Cette méthode utilise à la fois des raffinements successifs pour représenter le système à différents niveaux d'abstraction et des preuves mathématiques pour garantir la cohérence entre ces différents niveaux. SCADE [3] permet également de modéliser, vérifier, et exécuter des modèles de systèmes embarqués. Cette méthode applique des techniques de vérification formelle directement sur les modèles SCADE et un générateur de code certifié permet d'obtenir le code exécutable de l'application.

6 Conclusion

L'ingénierie dirigée par les modèles facilite le développement des systèmes embarqués complexes en permettant leur vérification et leur exécution sous forme de modèles. L'ap-

proche présentée dans cet article permet d'unifier la vérification et l'exécution embarquée de ces modèles en utilisant une seule définition de la sémantique du langage de modélisation. Cette définition de la sémantique est implémentée dans un interpréteur de modèles dédié à l'exécution de modèles pour l'embarqué. Le modèle peut également être vérifié en connectant des outils de V&V à cet interpréteur et en réutilisant la même sémantique opérationnelle que celle utilisée pour l'exécution.

Cette approche permet d'améliorer la qualité de développement des systèmes embarqués. Elle permet notamment d'assurer que les propriétés vérifiées pendant la phase de vérification le restent lors de l'exécution. Elle facilite également la compréhension des résultats de vérification et reste applicable dans le cas des langages semi-formels comme UML. L'utilisation de l'interpréteur de modèles EMI couplé au model-checker OBP2 a en effet permis de simuler, model-checker, monitorer, et exécuter des modèles UML. L'interpréteur de modèles EMI doit néanmoins encore être amélioré pour évaluer et mettre en oeuvre cette approche sur des systèmes embarqués industriels.

Remerciements Ce projet est partiellement financé par Davidson Consulting. L'auteur remercie particulièrement David Olivier, Matthias Brun, Ciprian Teodorov, Frédéric Jouault, et Philippe Dhaussy pour leurs conseils et commentaires avisés sur le projet.

Références

[1] Jean-Raymond Abrial. *Modeling in Event-B : System and Software Engineering*. Cambridge University Press, New York, NY, USA, 2013.

[2] Mojtaba Bagherzadeh, Nicolas Hili, and Juergen Dingel. Model-level, Platform-independent Debugging in the Context of the Model-driven Development of Real-time Systems. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, pages 419–430, New York, USA, 2017. ACM.

[3] Gérard Berry. SCADE : Synchronous Design and Validation of Embedded Control Software. In S. Ramesh and Prahladavaradan Sampath, editors, *Next Generation Design and Verification Methodologies for Distributed Embedded Control Systems*, pages 19–33, Dordrecht, 2007. Springer Netherlands.

[4] Valentin Besnard, Matthias Brun, Philippe Dhaussy, Frédéric Jouault, David Olivier, and Ciprian Teodorov. Towards one Model Interpreter for Both Design and Deployment. In *3rd International Workshop on Executable Modeling (EXE)*, Austin, United States, September 2017.

[5] Valentin Besnard, Matthias Brun, Frédéric Jouault, Ciprian Teodorov, and Philippe Dhaussy. Embedded UML Model Execution to Bridge the Gap Between Design and Runtime. In *MDE@DeRun 2018 : First International Workshop on Model-Driven Engineering for Design-Runtime Interaction in Complex Systems*, Toulouse, France, June 2018.

[6] Valentin Besnard, Matthias Brun, Frédéric Jouault, Ciprian Teodorov, and Philippe Dhaussy. Unified LTL Verification and Embedded Execution of UML Models. In *ACM/IEEE 21th International Conference on Model Driven Engineering Languages and Systems (MODELS '18)*, Copenhagen, Denmark, October 2018.

[7] Erwan Bousse, Thomas Degueule, Didier Vojtisek, Tanja Mayerhofer, Julien Deantoni, and Benoit Combemale. Execution Framework of the GEMOC Studio (Tool

Demo). In *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering*, SLE 2016, pages 84–89, New York, NY, USA, 2016. ACM.

[8] Asma Charfi Smaoui, Chokri Mraidha, and Pierre Boulet. An Optimized Compilation of UML State Machines. In *ISORC - 15th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*, Shenzhen, China, April 2012.

[9] Federico Ciccozzi. Unicomp : A Semantics-aware Model Compiler for Optimised Predictable Software. In *Proceedings of the 40th International Conference on Software Engineering : New Ideas and Emerging Results*, ICSE-NIER '18, pages 41–44, New York, NY, USA, 2018. ACM.

[10] Federico Ciccozzi, Ivano Malavolta, and Bran Selic. Execution of UML models : a systematic review of research and practice. *Software & Systems Modeling*, April 2018.

[11] Eran Gery, David Harel, and Eldad Palachi. Rhapsody : A Complete Life-Cycle Model-Based Development System. In Michael Butler, Luigia Petre, and Kaisa Sere, editors, *Integrated Formal Methods*, pages 1–10, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

[12] Padma Iyenghar, Elke Pulvermueller, Clemens Westerkamp, Juergen Wuebbelmann, and Michael Uelschen. *Model-Based Debugging of Embedded Software Systems*, pages 107–132. Springer New York, New York, NY, 2017.

[13] Agnes Lanusse, Yann Tanguy, Huascar Espinoza, Chokri Mraidha, Sebastien Gerard, Patrick Tessier, Remi Schnekenburger, Hubert Dubois, and François Terrier. Papyrus UML : an open source toolset for MDA. In *Proceedings of the Fifth European Conference on Model-Driven Architecture Foundations and Applications*, pages 1–4, 2009.

[14] Daniel Leroux, Martin Nally, and Kenneth Hussey. Rational Software Architect : A tool for domain-specific modeling. *IBM systems journal*, 45(3) :555–568, 2006.

[15] Xavier Leroy. The CompCert C verified compiler : Documentation and user's manual. Intern report, Inria, June 2017.

[16] Tanja Mayerhofer and Philip Langer. Moliz : A Model Execution Framework for UML Models. In *Proceedings of the 2nd International Master Class on Model-Driven Engineering : Modeling Wizards*, MW '12, pages 3 :1–3 :2, New York, NY, USA, 2012. ACM.

[17] OMG. Semantics of a Foundational Subset for Executable UML Models, October 2017. `https://www.omg.org/spec/FUML/1.3/PDF`.

[18] OMG. Unified Modeling Language, December 2017. `https://www.omg.org/spec/UML/2.5.1/PDF`.

[19] Sebastien Revol, Géry Delog, Arnaud Cuccurru, and Jérémie Tatibouët. Papyrus : Moka overview, 2018. `https://wiki.eclipse.org/Papyrus/UserGuide/ModelExecution`.

[20] Ciprian Teodorov, Philippe Dhaussy, and Luka Le Roux. Environment-driven reachability for timed systems. *International Journal on Software Tools for Technology Transfer*, 19(2) :229–245, April 2017.

[21] Ciprian Teodorov, Luka Le Roux, Zoé Drey, and Philippe Dhaussy. Past-Free[ze] reachability analysis : reaching further with DAG-directed exhaustive state-space analysis. *Software Testing, Verification and Reliability*, 26(7) :516–542, 2016.