Vérification d'un générateur de code par génération d'annotations

Arnaud Dieumegard
Arnaud.Dieumegard@enseeiht.fr

Marc Pantel
Marc.Pantel@enseeiht.fr

RÉSUMÉ. Dans le cadre de la vérification de systèmes critiques, cette communication présente une stratégie de vérification du générateur de code GENEAUTO qui traduit des modèles SIMULINK en langage C. Il s'agit de définir une méthodologie de vérification pouvant être exploitée par les experts métiers utilisateurs du générateur. Ceux-ci peuvent définir eux-même des blocs métiers pour leurs modèles dont le code généré doit être vérifié par rapport à la sémantique de ces blocs. La vérification du code généré doit donc s'appuyer sur des techniques relevant d'un niveau d'abstraction abordable par les futurs utilisateurs. Pour cela, les utilisateurs doivent fournir une spécification formelle de chacun de leurs blocs métiers SIMULINK. Celle-ci est ensuite utilisée pour générer des annotations sur le code (représentant le comportement des blocs) permettant la vérification. Cette stratégie s'appuie sur l'utilisation d'annotations ACSL vérifiées grâce à des solveurs SMT au sein de l'atelier FRAMA-C.

Mots clés : Systèmes critiques, vérification, génération de code, ACSL, solveurs SMT

1 Introduction

1.1 Contexte

Les générateurs automatiques de code (GAC) à partir de modèles sont utilisés dans le domaine des systèmes critiques pour réduire le temps de développement, et pour faciliter la vérification de la correction du code cible généré par rapport au modèle source considéré. L'évolution de la complexité de tels systèmes conduit à une augmentation des exigences en terme de sûreté au niveau des normes concernées (DO178/ED12, IEC61508, ISO26262, ECSS). La part prépondérante qu'occupe à présent le logiciel dans ces systèmes conduit à des coûts de développement et de maintenance élevés. En effet, la validation et la vérification sont principalement effectuées sur le système final et les anomalies détectées peuvent entraîner une remise en cause, non seulement de certaines exigences initiales, mais également des nombreux intermédiaires dans le développement qui en sont dérivés. L'ingénierie dirigée par les modèles permet la validation des exigences et la vérification des intermédiaires dans le développement en s'appuyant sur des modèles pour détecter les anomalies le plus tôt possible, et sur la génération automatique de code à partir de ces modèles pour réduire les risques d'introduction d'anomalies lors des phases de codage manuel. Ces approches sont de plus en plus utilisées dans le milieu industriel, citons l'exploitation de la SAO puis de SCADE

chez Airbus depuis plus de 20 ans, et celle de SIMULINK/STATEFLOW ou d'UML/-SysML/MARTE dans de nombreux développements actuels et dans la plupart des projets prévus pour un futur proche selon les équipes de Recherche et Développement de nos partenaires.

Les normes actuelles de certification des systèmes critiques permettent de s'appuyer sur les résultats de vérification effectuée par des outils automatiques si ces outils ont été eux-mêmes qualifiés, c'est-à-dire développés en respectant les même normes. Les anomalies d'un générateur automatique de code peuvent transformer une source correcte en une cible incorrecte (le programme C généré à partir d'un modèle, le binaire exécutable généré à partir d'un programme C, ...). Les normes considèrent le générateur et le système généré au même niveau de criticité par rapport aux activités de validation et vérification.

1.2 Motivations

Les normes de certification imposent une analyse de tous les risques potentiels, la définition d'activités de vérification associées à chaque risque et la traçabilité entre tous les éléments de développement. Pour les satisfaire, les approches classiques pour la qualification des générateurs de code reposent principalement sur le respect d'un processus de développement très détaillé pour identifier et couvrir tous les risques, l'écriture d'exigences très précises, la vérification par relecture indépendante et exécution de tests aux différentes étapes du processus (unitaire, intégration, fonctionnel, déploiement) avec une couverture très complète des exigences. Par exemple, la qualification du générateur de code KCG qui traduit en code C des modèles en SCADE [6] (langage de modélisation graphique semblable à SIMULINK) avec une sémantique synchrone issue de LUSTRE [8], exploite une vérification principalement basée sur une couverture de test de type MC/DC adaptée au langage OCAML [20, 21].

Cependant, le test ne peut pas être exhaustif et la relecture est une activité humaine peu efficace et extrêmement coûteuse. De plus, il est très difficile de construire des stratégies de test pertinentes pour un générateur de code. En effet, contrairement aux systèmes classiques pour lesquels il est généralement possible de définir un domaine et un co-domaine d'exploitation restreint (par exemple, en utilisant la programmation par contrat), il est difficile de prévoir toutes les utilisations possibles d'un langage, car celles-ci dépendent souvent du contexte (domaine, entreprise, ...), du problème modélisé, de l'utilisateur, des outils disponibles...

Les approches formelles permettent d'effectuer une vérification exhaustive pour certaines parties, ou certaines abstractions d'un système voire pour le système réel complet. Leur utilisation reste coûteuse mais de nombreuses expériences sur des cas d'études industriels réalistes ont mis en évidence leurs pertinences et les gains possibles par rapport aux tests dans le cadre des systèmes critiques (citons par exemple la vérification de modèles, l'analyse statique pour la détection des erreurs d'exécution [25], ou d'estimation de propriétés quantitatives [26], le développement certifié pour la correction par rapport aux exigences utilisateurs utilisant la méthode B [13], ou CAVEAT [23]). Les premières exploitation dans des projets réels (ALSTOM pour Météor, Airbus SAS pour l'A380, ...) ont conduit à des évolutions des normes pour prendre en compte ces nouvelles approches de la V & V. La version C du standard aéronautique DO178/ED12 qui doit être finalisée en 2010 contient des annexes spécifiques pour l'exploitation de méthodes formelles, le développement à base de modèles et la qualification des outils de génération et de vérification. Plusieurs problèmes doivent encore être traités par la communauté académique [16] mais une exploitation à moyen terme pour

la construction des outils de développement de systèmes industriels complets semble maintenant envisageable.

1.3 Application

Les travaux présentés dans cet article exploitent l'outil de vérification par analyse statique Frama-C [1] pour vérifier le code C généré par GENEAUTO [2] par rapport à la sémantique des blocs élémentaires de SIMULINK. Ces travaux, viennent enrichir les expériences réalisées avec l'assistant de preuve Coq réalisés dans le projet ITEA éponyme entre 2006 et 2008 qui avaient pour objectif la spécification et l'implantation d'un générateur de code qualifiable qui produit du code embarqué critique certifiable en langage C à partir de modèles définis avec un sous-ensemble de SIMULINK/STATEFLOW et SCICOS. Les partenaires industriels du projet, issus des domaines aéronautique, automobile et spatial, conseillés par les partenaires académiques, ont défini le cahier des charges et les spécifications détaillées. Ceux-ci comportent deux éléments : d'une part, un sous-ensemble discret de SIMULINK/STATEFLOW et SCICOS adapté à la modélisation d'applications critiques de contrôle/commande, en portant une attention particulière à des restrictions syntaxiques permettant d'assurer, par construction, de bonnes propriétés au niveau des modèles ; d'autre part, l'architecture d'un générateur de code qui permet d'élargir simplement le spectre du langage source [28, 27], et la spécification détaillée de chaque composant. Un point important dans le projet était l'expérimentation de méthodes formelles [11, 10, 9] pour la vérification de l'implémentation des composants par rapport à leur spécification classique et les échanges avec les autorités de certification autour du processus proposé.

Notre contribution se place dans le cadre de la vérification formelle de la génération automatique proposée dans GENEAUTO. En collaboration avec les experts industriels de la qualification, nous proposons un processus de vérification combinant approches formelles et classiques. Dans ces travaux, une partie de la spécification des blocs de contrôle SIMULINK écrite en langage naturel est transcrite sous la forme d'une description formelle des entrées/sorties des blocs. Cette description est ensuite traduite dans un langage d'annotations qui allié a un moteur de plus faible préconditions permet la vérification du code à l'aide de solveur SMT dans l'outil FRAMA-C.

Cet article présente d'une part le résultat de l'analyse effectuée pour le choix de formalisme de spécification et vérification formelle exploités pour une première expérience dans le projet ; et d'autre part certaines leçons tirées de cette expérience et les choix effectués dans une seconde expérience afin de décrire le comportement des blocs ainsi que les méthodes formelles utilisées pour le vérifier.

L'article est organisé de la façon suivante. La section 2 présente les principales techniques de vérification formelles appliquées à des générateurs de code et justifie le choix des assistants de preuve pour GENEAUTO. La section 3 décrit SIMULINK ainsi que les blocs que nous utilisons pour cette étude. La section 4 présente les différents outils de vérification que nous utilisons. Dans la section 5, nous introduisons le générateur de code GENEAUTO ainsi que les résultats préliminaires que nous avons obtenu autour d'une expérimentation. Enfin, les conclusions et perspectives sont tracées dans la dernière section.

1. `http://frama-c.com/index.html`
2. `http://www.geneauto.org`

2 Sélection d'une technique de vérification formelle

Dans la première phase de GENEAUTO, nous avons étudié l'approche formelle la mieux adaptée au développement d'un générateur qui devra être qualifié. Plusieurs critères ont été considérés liés aux coûts d'acquisition, d'exploitation et de maintenance de ces technologies, aux coûts annexes liés à son utilisation (i.e. le développement et la qualification d'outils supplémentaires) et aux risques qu'elle ne détecte pas toutes les anomalies. Deux critères se sont révélés décisifs : la réduction du risque d'anomalies résiduelles après vérification ; et la durée de la phase de vérification lors de l'utilisation du générateur. D'une part, vis-à-vis des autorités de certification, il faut choisir la technologie qui minimise les risques pour le système final, et donc qui assure que le plus grand nombre d'anomalies (voire l'intégralité) sont éliminées. D'autre part, lors de la détection d'une anomalie sur le système en service, celle-ci doit être corrigée et une nouvelle version du système doit être déployée le plus rapidement possible pour réduire les coûts d'immobilisation et les risques si le service n'est pas suspendu. Dans le domaine de l'aéronautique, le délai requis est usuellement d'une semaine de la signalisation de l'anomalie au déploiement de la version corrigée du système.

Vérification de la cible (Translation validation)

Cette technique, introduite par [22] dans le cadre des projets européens SAFEAIR, propose de comparer, à chaque utilisation du générateur, la source et la cible en s'appuyant sur des méthodes adaptées à la propriété de correction souhaitée. Cette comparaison peut être syntaxique (par exemple, une classe JAVA est produite pour chaque classe UML) ou sémantique (le résultat de l'exécution est le même pour la sémantique de la source et de la cible). En pratique, le générateur est complété par un module qui vérifie statiquement les propriétés de correction par une analyse de la correspondance entre source et cible. Cette vérification peut s'appuyer sur différentes techniques classiques comme la vérification de modèles [22, 31] et l'interprétation abstraite [24], ou sur des algorithmes spécifiques à la propriété souhaitée [3, 29].

Cette approche a un avantage majeur : le générateur est une boîte noire qui n'est pas contrainte par la technique de vérification si ce n'est d'établir quelques liens de traçabilité entre source et cible pour simplifier la vérification (par exemple, noms des variables et des fonctions entre langage C et assembleur).

L'utilisation de vérifieur de modèles a été expérimentée sur plusieurs systèmes industriels réels dans SAFEAIR pour la traduction de modèles synchrones en langage C, puis de langage C en assembleur, mais elle a un coût de vérification encore trop élevé pour passer à l'échelle. Par contre, l'utilisation de l'interprétation abstraite est très prometteuse et permet de vérifier le cas de plusieurs centaines de milliers de lignes de code en C. Les techniques spécifiques à certaines propriétés ont en général un objectif restreint et passent très bien à l'échelle.

Toutefois, elle présente un défaut majeur. Quand le module de vérification échoue, l'utilisateur ne dispose que de peu d'informations sur l'origine de l'échec (l'anomalie peut être liée au générateur ou au vérifieur). Il est ensuite nécessaire de faire une analyse longue et coûteuse incompatible avec la durée maximum d'un cycle de maintenance. Pour éviter ce problème, il faut qualifier le module de vérification lui-même avec une technologie différente. En ce qui concerne les techniques spécifiques, elles peuvent être développées avec un assistant de preuve tel COQ [29] ou ISABELLE [3] et obtiennent de bons résultats pour des problèmes bien délimités. Par contre, les résultats préliminaires de vérification par des assistants de preuve d'outils de vérification de

modèle [30] ou d'interprétation abstraite [4] ne permettent pas, à notre connaissance, et suite à des discussions avec les développeurs de ces technologies [12], d'envisager un passage à l'échelle pour l'instant. Plus précisément, la vérification de la vérification de modèles est réalisable mais cette utilisation particulière de cette technique n'est pas encore transposable à des systèmes réels ; l'interprétation abstraite permet de traiter des modèles réalistes mais son implémentation efficace n'est pas encore vérifiable mécaniquement en un temps raisonnable. Une troisième possibilité est l'utilisation des techniques déductives dérivées de la logique de Hoare. Celles-ci ont été expérimentées avec succès à travers l'outil CAVEAT dans le cadre de l'A380 et se poursuive avec l'outil frama-C dont la qualification partielle ou totale est en cours d'évaluation dans le cadre du projet ITEA2 OPEES. L'utilisateur peut annoter complétement les intermédiaires de développement et fournir ainsi la preuve complète de correction du code C considéré par rapport aux exigences exprimées sous la forme d'annotation (pre-conditions, post-conditions et invariants). En cas d'échec dans la vérification, il s'agit généralement d'un manque de ressource en calcul ou mémoire pour conclure en un temps raisonnable. L'utilisateur peut alors compléter l'annotation du code C pour atteindre une forme dont la vérification réussi.

Code auto-certifié (Proof carrying code)

Cette technique proposée par [19] consiste à produire en plus de la cible générée, la preuve que cette cible est correcte par rapport à la source. Il s'agit donc également d'une vérification à chaque utilisation. Cette preuve peut être produite par l'utilisateur ou par le générateur. Elle peut être incorrecte et devra donc être vérifiée indépendamment lors de l'utilisation de la cible. Il s'agit donc d'une approche semblable à la vérification de la cible. Cette technique a été principalement appliquée aux systèmes nécessitant une garantie de sécurité, par exemple dans le cas de code mobile. Par rapport à la vérification de la cible, d'une part, il peut être plus coûteux de produire une preuve qu'une propriété est préservée que de vérifier que cette propriété est satisfaite par la cible avec une autre technique (en particulier pour la vérification de modèles ou l'interprétation abstraite qui s'appuient sur une exécution symbolique exhaustive relativement éloignée des mécanismes de preuve) ; d'autre part, le générateur doit être une boîte blanche pour permettre la construction de la preuve conjointement à la génération du code. Par contre, la vérification de la preuve est une activité bien maîtrisée qui passe à l'échelle d'exemples réels. Le problème essentiel reste donc la possibilité d'un échec de vérification, l'analyse et la correction de ce type d'anomalie lors de la maintenance. La complexité de la conception des patrons de preuve pour la génération est souvent du même ordre que celle de la construction de la preuve de correction du générateur lui-même. De plus, le générateur doit être une boîte blanche donc l'avantage boîte noire de la vérification de la cible est perdu. Le code auto-certifié ne présente pas d'avantages majeurs par rapport au développement prouvé dans notre contexte. En fait, dans le cadre de la sécurité, la vérification de la preuve permettait également d'assurer que la cible n'avait pas été modifiée pour le code mobile.

Développement prouvé et Preuve assistée

La vérification est appliquée au générateur lui même. Il s'agit de spécifier formellement les exigences du générateur, son implantation et la preuve de correction de cette implantation en exploitant un outil de dérivation de programmes correct à partir d'une

spécification formelle, ou un assistant de preuve. Cette technique, la première expérimentée dans cet objectif, a donné de nombreux résultats concluants. Nous pouvons citer les travaux initiaux [18] avec LCF et la bibliographie de [5]. Parmi les travaux les plus récents, le projet COMPCERT [3] exploite COQ [4] pour spécifier et vérifier un générateur de code d'un sous-ensemble du langage C vers du code binaire POWERPC optimisé [14, 15]. Pour simplifier cette vérification, le générateur a été décomposé en transformations successives, chaque transformation est prouvée correcte en utilisant COQ ou vérifiée par un outil complémentaire développé lui même en utilisant COQ.

Cette approche permet d'assurer la préservation de propriétés sémantiques au niveau de la cible qui ont préalablement été spécifiées sur le source. Elle est très prometteuse dans le domaine de la vérification formelle et notamment dans les applications des systèmes embarqués critiques. Elle passe à l'échelle de systèmes industriels réels. Elle ne nécessite aucune vérification pouvant échouer lors de son application, ou plus précisément, la vérification de la cible est exploitée pour des optimisations du code qui peuvent être désactivées, ou s'appuyer sur des algorithmes plus simples et moins performants mais qui sont vérifiés au niveau du générateur lui-même (par exemple, l'allocation de registres). Enfin, aucune anomalie ne peut rester au niveau du générateur par rapport aux exigences explicites car il s'agit de preuves exhaustives. Il faut noter, par contre, que les coûts d'acquisition, d'exploitation et de maintenance pour ces technologies de preuve assistée sont particulièrement élevés par rapport à un développement classique, même en intégrant le coût des tests actuels. De plus, il est actuellement difficile de trouver en dehors du milieu académique des équipes de développement prêtes à se lancer dans de telles technologies. En effet, il s'agit d'un marché de niche (citons par exemple le cas de GemAlto et Trusted Logic dans le domaine de la sécurité sur cartes à puces) pour lequel nous ne connaissons pas de partenaires industriels disponibles dans le contexte des générateurs de code. Enfin, il faut disposer d'outils de vérification de preuves et d'extraction de code exécutable depuis les preuves qui soit eux mêmes qualifiés. Sur ce dernier point, il devrait être possible de s'appuyer à moyen terme sur une implantation d'un sous-ensemble de COQ en COQ [5] pour la vérification [1] et l'extraction [17, 7].

Synthèse

Dans le cadre de GENEAUTO, deux critères ont prévalu : d'une part, la possibilité d'éliminer toutes les anomalies du générateur par rapport à ses exigences (il serait difficile de justifier auprès des autorités de certification l'utilisation d'une technologie pouvant laisser des anomalies sachant qu'une autre technologie permet de les éliminer) ; et d'autre part, la réduction des coûts de vérification à la génération qui permet de mieux maîtriser la durée des opérations de maintenance des systèmes.

Nous avons expérimenté dans une première étape le développement des outils élémentaires en exploitant l'assistant de preuve Coq. Ceci a donné de très bons résultats lorsque le développement des outils est indépendant des utilisateurs finaux comme dans le cas de l'ordonnanceur des blocs [9]. Par contre, lorsque les utilisateurs finaux doivent pouvoir intervenir dans la spécification et la vérification, le coût d'accessibilité des technologies de preuve assistée ne permet pas une coopération raisonnable entre les partenaires industriels et académiques. Pour faciliter la lecture et l'écriture des aspects

3. `http://compcert.inria.fr`
4. `http://coq.inria.fr`
5. `http://www.lix.polytechnique.fr/~barras/coq-implicit`

sémantiques, nous avons choisi pour une seconde expérience d'expérimenter la vérification de la cible en nous appuyant sur des techniques dérivées de la logique de Hoare pour lesquelles l'utilisateur pourra intervenir plus facilement en cas d'échec dans la vérification, par rapport à la vérification de modèles ou l'interprétation abstraite.

Suite à ces différents retours d'expériences issus du projet GENEAUTO, aux gains de maturité des outils de vérification dérivé de la logique de Hoare et des perspectives de qualification de ces outils, nous avons choisi de nous orienter vers l'utilisation de solveurs SMT pour prouver le code généré.

Nous nous focalisons dans cet article sur les formalismes utilisés pour spécifier les bibliothèques de blocs en essayant de les rendre exploitable par des utilisateurs finaux experts et leur utilisation à des fins de vérification.

3 Brève description de SIMULINK

SIMULINK est un environnement de modélisation et de simulation de systèmes adaptés aux algorithmes de commande et contrôle qui jouent un rôle essentiel dans la partie critique des systèmes embarqués, par exemple les commandes de vol électriques, le pilotage automatique, l'injection électronique, le freinage, la régulation de vitesse, etc. Ce langage s'est imposé comme standard de fait pour la conception de ce type de systèmes.

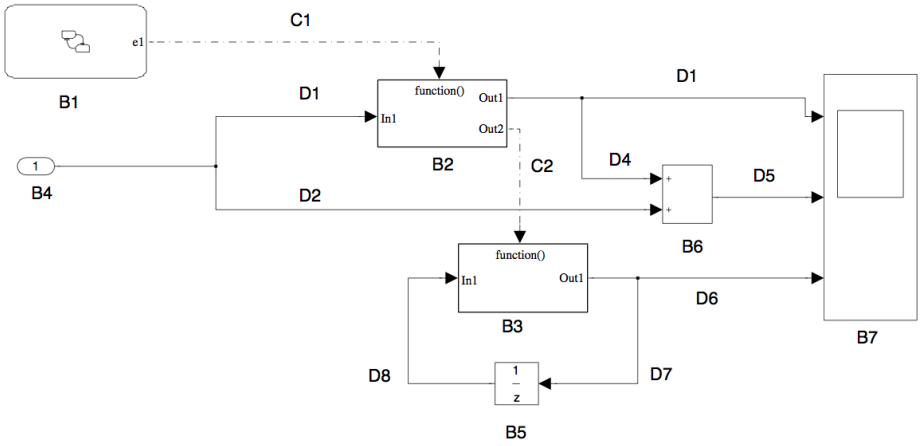


FIGURE 1 – Exemple de diagramme SIMULINK

Un modèle est représenté graphiquement par des diagrammes composés de nœuds reliés par des arcs (voir Fig. 1). Les nœuds représentent des fonctions et les arcs qui les relient représentent des signaux qui peuvent être des flots de données (transportant des valeurs) ou de contrôle (transportant des événements pour provoquer immédiatement l'exécution des sous-systèmes cible). Nous nous limitons dans GENEAUTO aux modèles discrets. Pour plus de détails, le lecteur peut se référer au site du fournisseur de l'outil [6].

On donne ci dessous la représentation graphique des blocs SIMULINK utilisés dans cette étude (Figure 2). Ces trois blocs ont été choisis en fonction de leur intérêt et de leur représentativité des structures de contrôle du langage C :

6. http://www.mathworks.com

– Bloc Sum : Les valeurs des éléments des ports d'entrées du bloc sont sommés ou soustraits pour calculer la valeur de sortie (opérations arithmétiques, boucles imbriquées).
– Bloc DelayRE : Ce bloc est fonctionnel. Il permet, en fonction des valeurs des booléens E et R (Reset) d'affecter aux sorties les valeurs d'entrée. Si R est activé alors les sorties (y et x_out) sont réinitialisées à la valeur initiale (IV), autrement y prend la valeur de x_In et x_out dépend de E (x_out prend la valeur u ou x_In). Il s'agit d'une composition de deux structures if/then/else.
– Bloc MultiPortSwitch : Ce bloc permet d'effectuer un traitement de type switch/-case. La première entrée permet de définir laquelle des autres entrées est utilisée comme valeur de sortie.

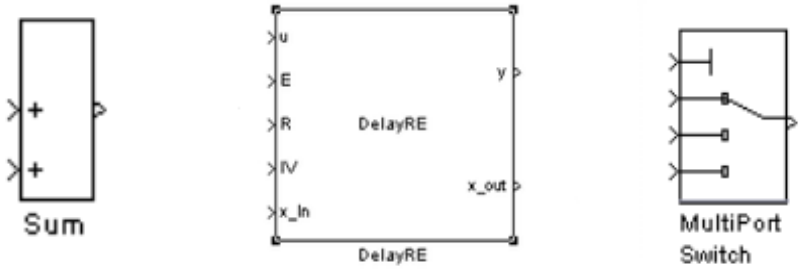


FIGURE 2 – représentation graphique de certains blocs SIMULINK

4 Outils de vérification déductive / analyse statique

La vérification déductive exploite des axiomes et des règles de déduction pour construire la preuve de correction d'un système. Les travaux résumés dans cet article exploitent le langage ACSL [7] pour écrire les propriétés souhaitées par des annotations sur des programmes C. L'analyse statique du code source permet de calculer des informations au sujet du code source d'un programme sans exécuter celui-ci.

4.1 ACSL

Le langage de spécification de propriétés ACSL est un langage de spécification comportementale pour les programmes en langage C. Sa sémantique est proche de celle de JML [8] et de l'analyseur de code source CADUCEUS [9] du projet WHY [10]. Les annotations ACSL se présentent sous la forme de commentaires du code C dont les caractères de début (// ou /*) sont postfixés par le caractère @.

Malgré le fait que ces annotations se situent dans des commentaires, ACSL reste un langage formel et peut donc à ce titre être utilisé par des programmes externes (ces annotations ne gênent donc en rien la compilation du programme d'origine).

7. ANSI/ISO C Specification Language
8. http://www.cs.ucf.edu/~leavens/JML/
9. http://caduceus.lri.fr/
10. http://why.lri.fr/

Nous décrirons dans la section 5 une partie des annotations qui composent ce langage. Pour plus d'informations, voir le document de référence [11].

4.2 L'outil FRAMA-C

L'outil FRAMA-C est un atelier logiciel composé de greffons (plugins) permettant différents types d'analyses sur le code source. Ces analyses s'appuient sur de l'analyse statique et extraient du code des informations de métrique (profondeur de structures de contrôle, nombre d'allocations statiques, ...), des informations sur l'origine des valeurs des variables, recherchent du code mort (une liste plus détaillée des fonctionnalités peut être trouvée sur le site de l'éditeur cité précédemment).

FRAMA-C a pour vocation d'être un outil *correct* et *complet* vis-a-vis de sa spécification. A ce titre, certaines de ces extensions permettent de faire de la vérification formelle sur le code analysé.

FRAMA-C nous permet d'effectuer une vérification déductive du code généré par l'outil GENEAUTO par le biais du greffon WP [12].

4.2.1 Le greffon WP

L'extension WP implante un calcul de plus faible préconditions en se basant sur les annotations ACSL et le code C du programme. Elle permet pour chaque annotation de générer des obligations de preuve qui sont ensuite vérifiés au sein d'assistants de preuve comme COQ, ISABELLE ou PVS, ou des outils automatiques supportés par l'outil Why (ALT-ERGO, Z3, CVC3, SIMPLIFY, ...).

Grace à cette extension et à l'utilisation des prouveurs de théorèmes automatisés, on s'abstrait totalement de l'utilisation d'outil tels que COQ dont l'utilisation reste difficile pour les utilisateurs ciblés.

5 Intégration des approches formelles dans GENEAUTO

L'intégration d'approches formelles dans un processus industriel classique pour le développement d'outils qualifiés est une des innovations de GENEAUTO. Il s'agit de prendre en compte les contraintes industrielles sous leur forme habituelle (exigences en langage naturel, langages de modélisation utilisés, interopérabilité des composants au sein d'une chaîne d'outils, normes de certification) et de proposer une approche basée sur la vérification par analyse statique.

Cela nous a conduit à suivre des approches différentes des solutions habituelles. Par exemple, la plupart des approches formelles s'appuient sur une définition de la sémantique des langages source et cible et sur la preuve de préservation de cette sémantique. [2] s'intéressent aux événements d'entrée/sortie avec l'environnement du programme et montrent que la cible en assembleur va effectuer les mêmes séquences d'entrée/sortie que le source en C. [22, 24] considèrent également les modifications de la mémoire effectuées par les programmes.

11. `http://frama-c.com/download/acsl_1.5.pdf`
12. Weakest Preconditions

5.1 Sélection du cas d'étude : la génération de code pour blocs élémentaires

Au sein de chaque entreprise, chaque projet repose sur une bibliothèque de blocs spécifique. Exploiter une preuve de correction sémantique avec un assistant de preuve demande donc de spécifier dans le langage de cet assistant (GALLINA pour COQ par exemple) les exigences pour la bibliothèque de blocs puis de prouver que le code généré préserve ces exigences. Nous proposons une approche qui évite cette étape et s'appuie sur des langages de spécification plus proche des activités de programmation classiques.

Pour simplifier les activités de spécification des exigences, de développement et favoriser les extensions, l'architecture choisie pour le générateur est modulaire. Elle est composée d'une succession d'étapes de transformations et de vérification qui conduisent du système représenté par un modèle vers du code C. Une version simplifiée de cette architecture est présentée dans la figure 3.

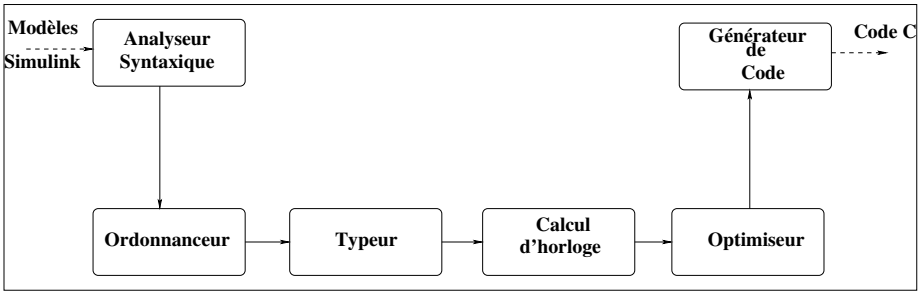


FIGURE 3 – Architecture du générateur de code GENEAUTO

Dans cet article, nous ne détaillerons pas les différents composants du générateur. Une vision synthétique est disponible dans [28, 27]. La description complète est accessible sur le site du projet.

Afin d'illustrer notre propos, nous allons nous appuyer sur la description de trois blocs SIMULINK élémentaires dont nous avons donné la représentation en figure 2.

Spécification habituelle des blocs

Pour chacun de ces blocs a été défini durant le projet GENEAUTO une spécification en langage naturel (e.g Specification en langage naturel 1 pour le bloc Sum). Cette spécification servira de point d'entrée à la formalisation au sein de notre approche.

Specification en langage naturel 1

Inputs

Unlimited number of inputs, but must be of the same data type. In case of different signal types, there can be a mix between scalar and vector signals in which case the scalar value will be summed to each element of the vector, or scalar and matrix signals in which case the scalar value will be summed to each element of the matrix but IMPOSSIBLE to mix vector and matrix signals. Vector (resp. matrix) input signals must have the same dimension. If only one input is given then it must be of scalar or vector signal type.

Output

Single output, of the same signal type and data type as the inputs having as value :

- *If scalar : the sum of its scalar inputs or the sum of its vector input elements*
- *If vector : the sum vector of its vector inputs, element by element (element-wise)*
- *If matrix : the sum matrix of its matrix inputs, element by element (element-wise)*

Spécification formelle des blocs

Nous avons traduit dans un formalisme mathématique les entrées/sorties de chacun de ces blocs, puis défini les types de données et de structures autorisés pour ce bloc ainsi que le formalisme de la sortie du bloc. Ceci constitue l'ensemble des pré/post conditions nécessaires à la définition comportementale du bloc (cf : Formalisation 1 pour le bloc Sum).

Formalisation 1

- $\mathbb{T}$ *the data type of Input elements*
- $\mathcal{I} = \{X_i\}$ *the set of input signals*
- $n \in \mathbb{N}$ *the number of input signals*
- $d, e \in \mathbb{N}$ *the dimensions of input signals (only d is necessary if input are vectors. d and e are necessary if inputs are matrixes).*

Input must be one of the following :

1. $\forall i \in [1, n], \mathcal{I} = \{X_i \in \mathbb{T}\}$
2. $\forall i \in [1, d], a_i \in \mathbb{T}, \mathcal{I} = \left\{ X = \begin{pmatrix} a_1 \\ \vdots \\ a_d \end{pmatrix} \right\}$
3. $n > 1, \forall i \in [1, n], \exists j \in \{1, d\}, \exists k \in [1, j], a_{i,k} \in \mathbb{T}, \mathcal{I} = \left\{ X_i = \begin{pmatrix} a_{i,1} \\ \vdots \\ a_{i,j} \end{pmatrix} \right\}$
4. $n > 1, \forall i \in [1, n], \exists j \in \{1, d\}, \exists k \in \{1, e\}, \forall l \in [1, j], \forall m \in [1, k], a_{i,l,m} \in \mathbb{T}, \mathcal{I} = \left\{ X_i = \begin{pmatrix} a_{i,1,1} & \cdots & a_{i,1,k} \\ \vdots & \ddots & \vdots \\ a_{i,j,1} & \cdots & a_{i,j,k} \end{pmatrix} \right\} \wedge (d = 1) \Leftrightarrow (e = 1)$

Assuming :

- S *is the output signal*

According to the input formalization, here are the output formalizations :

1. $S = \{\forall i \in [1, n], \exists \triangle_i \in \{+, -\}, \sum_{i=1}^{n} \triangle_i X_i\}$
2. $S = \{\forall i \in [1, d], \exists \triangle_i \in \{+, -\}, \sum_{i=1}^{d} \triangle_i a_i\}$
3. $S = \{\forall i \in [1, n], \exists \triangle_i \in \{+, -\}, \sum_{i=1}^{n} \triangle_i X_i\}$
4. $S = \{\forall i \in [1, n], \exists \triangle_i \in \{+, -\}, \sum_{i=1}^{n} \triangle_i X_i\}$

Concerning the 3 and 4, we force that :

$$\forall \triangle \in \{+, -\}, \forall i \in \{1, d\} \wedge \forall j \in \{1, e\}, \{s, a_{i,j}\} \in \mathbb{T}, X = \begin{pmatrix} a_{1,1} & \cdots & a_{1,j} \\ \vdots & \ddots & \vdots \\ a_{i,1} & \cdots & a_{i,j} \end{pmatrix},$$

$$s \triangle X = \begin{pmatrix} s \triangle a_{1,1} & \cdots & s \triangle a_{1,j} \\ \vdots & \ddots & \vdots \\ s \triangle a_{i,1} & \cdots & s \triangle a_{i,j} \end{pmatrix}$$

Le type de structures de données choisi comme entrée du bloc lors de la conception du système étant variable (scalaires, vecteurs, matrices), il à été nécessaire de définir des pré/post conditions adaptées à chaque type de structures.

Annotation du code

Conformément à ce qui a été défini dans la spécification formelle du bloc, nous souhaitons désormais intégrer au code généré les annotations ACSL nécessaires à la

vérification du code par FRAMA-C. Voici les exemples de code annoté correspondant au bloc Sum dans le Listing 1.

Listing 1

– *If scalar :*

```
/*@ requires \valid(&%o1) && \valid(&%i1) && ... ;
    requires \typeof(%o1) == \typeof(%i1) == ... ;
    ensures %o1 == %i1 + %i2 + ...;
*/
%o1 = %i1 + %i2 + ... ;
```

– *If one input as vector :*

```
/*@ requires \valid(&%o1) && \forall integer m;
      0 <= m < vector_size ==>  \valid(&%i1+m) ;
    requires \forall integer m; 0 <= m < vector_size ==>
      \typeof(%o1) == \typeof(%i1[m]) ;
    ensures %o1 == %i1[0] + %i1[1] + ...;
*/
%o1 = %i1[0] + %i1[1] + ... ;
```

– *If vector :*

```
/*@ requires \forall integer m;
      0 <= m < vector_size ==>
        \valid(&%o1+m) &&
        \valid(&%i1+m) && ... ;
    requires \forall integer m; 0 <= m < vector_size ==>
      \typeof(%o1[m]) == \typeof(%i1[m]) == ... ;
    ensures \forall integer m; 0 <= m < vector_size ==>
      %o1[m] == %i1[m] + %i2[m] + ... + %i5 + ... ;
*/
/*@ loop invariant 0 <= index <= vector_size;
    loop invariant \forall integer m; index <= m < vector_size ==>
          %o1[m] == \at(%o1[m],Pre);
    loop invariant \forall integer m; 0 <= m < index ==>
          %o1[m] == %i1[m] + %i2[m] + ... + %i5 + ...;
    loop variant vector_size - index;
*/
for (int index = 0; index < vector_size; index++){
  %o1[index] = %i1[index] + %i2[index] + ... + %i5 + ...;
}
```

– *If matrix :*

```
/*@ requires \forall integer m,n;
      0 <= m < matrix_size_x && 0 <= n < matrix_size_y ==>
        \valid(&%o1[m]+n) &&
        \valid(&%i1[m]+n) && ... ;
    requires \forall integer m,n;
      0 <= m < matrix_size_x && 0 <= n < matrix_size_y ==>
        \typeof(%o1[m][n]) == \typeof(%i1[m][n]) == ... ;
    ensures \forall integer m,n;
      0 <= m < matrix_size_x && 0 <= n < matrix_size_y ==>
        %o1[m][n] == %i1[m][n] + %i2[m][n] + ... + %i5 + ...;
*/
/*@ loop invariant 0 <= index_x <= matrix_size_x;
    loop invariant \forall integer m,n;
      0 <= m < index_x && 0 <= n < matrix_size_y ==>
        %o1[m][n] == %i1[m][n] + %i2[m][n] + ... + %i5 + ...;
    loop variant matrix_size_x - index_x;
*/
for (int index_x=0; index_x < matrix_size_x; index_x++) {
```

```
    /*@ loop invariant 0 <= index_y <= matrix_size_y;
        loop invariant index_x == \at(index_x,Here);
        loop invariant \forall integer n; 0 <= n < index_y ==>
          %o1[index_x][n] == %i1[index_x][n] + %i2[index_x][n] + ... + %i5 + ...;
        loop invariant \forall integer m,n;
          0 <= m < index_x && 0 <= n < matrix_size_y ==>
            %o1[m][n] == %i1[m][n] + %i2[m][n] + ... + %i5 + ...;
        loop variant matrix_size_y - index_y;
    */
    for (int index_y = 0; index_y < matrix_size_y; index_y++){
      %o1[index_x][index_y] = %i1[index_x][index_y] +
                %i2[index_x][index_y] + ... + %i5 + ...;
    }
  }
```

Nous avons choisi dans les deux derniers cas (addition de vecteurs ou de matrices), l'entrée $i5$ de type scalaire.

Dans les listings les ... sont à remplacer par les autres entrées du bloc.

Correspondance annotations/specification formelle

Les annotations présentées dans le Listing 1 peuvent être classées selon deux catégories :

- Les annotations correspondant aux pré/post conditions définies dans la spécification formelle :
  Elles font référence à la fonctionnalité attendue du bloc, dans le cas du bloc Sum, il s'agit des annotations utilisant la quantification universelle *forall* présentes dans les annotations *loop invariant* (dans les deux derniers exemples où les entrées sont des vecteurs ou des matrices). Dans le cas présent, les invariants de boucle nous assurent que les post conditions définies dans la spécification sont respectées. Dans les exemples, les post conditions sont indiquées grâce à l'annotation *ensure*. Les pré-conditions sont annoté avec le mot clé *requires* au début du bloc.
- Les annotations spécifiques à l'algorithme généré par GENEAUTO :
  Ce sont les autres annotations des exemples du Listing 1. Elles permettent dans les deux derniers cas de s'assurer de la terminaison des boucles et donc du programme (loop variant).

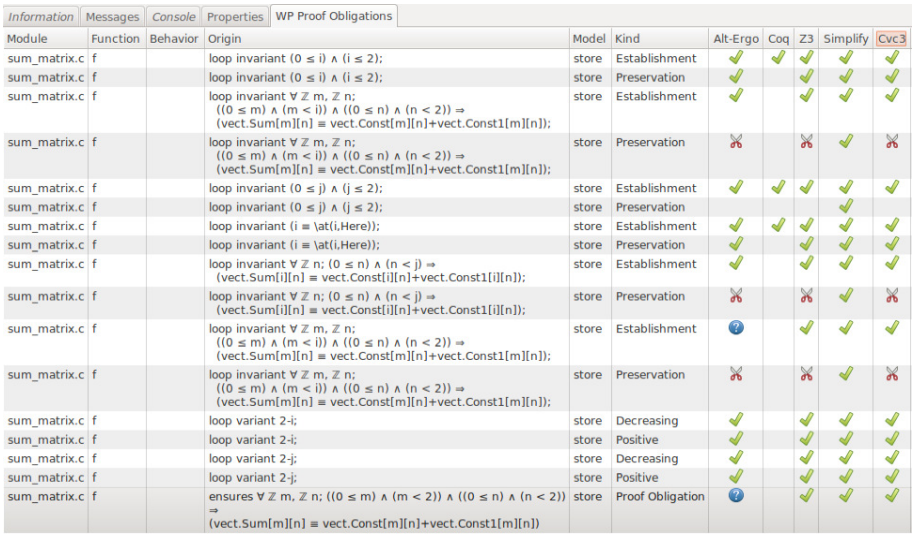
Vérification du code annoté

L'utilisation de l'outil FRAMA-C et de son extension WP nous permet désormais de générer la preuve de la correction de notre programme. On peut voir sur la figure 4 le résultat de la vérification.

Sur l'affichage précédent de l'outil FRAMA-C, chaque annotation est détaillée, et le résultat de la preuve (en fonction du solveur utilisé) est affiché en correspondance.

Bilan

L'exemple du bloc Sum a montré qu'une transformation d'une spécification formelle en un jeu d'annotations est possible au travers de l'utilisation d'ACSL et que la preuve du programme ainsi généré en est facilitée. Ceci repose sur le fait d'associer à chaque bloc SIMULINK un patron de code pour la génération auquel certaines annotations sont associées automatiquement.



Information | Messages | Console | Properties | WP Proof Obligations

Module	Function	Behavior	Origin	Model	Kind	Alt-Ergo	Coq	Z3	Simplify	Cvc3
sum_matrix.c	f		loop invariant (0 ≤ i) ∧ (i ≤ 2);	store	Establishment	✓	✓	✓	✓	✓
sum_matrix.c	f		loop invariant (0 ≤ i) ∧ (i ≤ 2);	store	Preservation	✓		✓	✓	✓
sum_matrix.c	f		loop invariant ∀ ℤ m, ℤ n; ((0 ≤ m) ∧ (m < i)) ∧ ((0 ≤ n) ∧ (n < 2)) ⇒ (vect.Sum[m][n] ≡ vect.Const[m][n]+vect.Const1[m][n]);	store	Establishment	✓		✓	✓	✓
sum_matrix.c	f		loop invariant ∀ ℤ m, ℤ n; ((0 ≤ m) ∧ (m < i)) ∧ ((0 ≤ n) ∧ (n < 2)) ⇒ (vect.Sum[m][n] ≡ vect.Const[m][n]+vect.Const1[m][n]);	store	Preservation	✗		✗	✓	✗
sum_matrix.c	f		loop invariant (0 ≤ j) ∧ (j ≤ 2);	store	Establishment	✓	✓	✓	✓	✓
sum_matrix.c	f		loop invariant (0 ≤ j) ∧ (j ≤ 2);	store	Preservation				✓	
sum_matrix.c	f		loop invariant (i ≡ \at(i,Here));	store	Establishment	✓	✓	✓	✓	✓
sum_matrix.c	f		loop invariant (i ≡ \at(i,Here));	store	Preservation	✓		✓	✓	✓
sum_matrix.c	f		loop invariant ∀ ℤ n; (0 ≤ n) ∧ (n < j) ⇒ (vect.Sum[i][n] ≡ vect.Const[i][n]+vect.Const1[i][n]);	store	Establishment	✓		✓	✓	✓
sum_matrix.c	f		loop invariant ∀ ℤ n; (0 ≤ n) ∧ (n < j) ⇒ (vect.Sum[i][n] ≡ vect.Const[i][n]+vect.Const1[i][n]);	store	Preservation	✗		✗	✓	✗
sum_matrix.c	f		loop invariant ∀ ℤ m, ℤ n; ((0 ≤ m) ∧ (m < i)) ∧ ((0 ≤ n) ∧ (n < 2)) ⇒ (vect.Sum[m][n] ≡ vect.Const[m][n]+vect.Const1[m][n]);	store	Establishment	?		✓	✓	✓
sum_matrix.c	f		loop invariant ∀ ℤ m, ℤ n; ((0 ≤ m) ∧ (m < i)) ∧ ((0 ≤ n) ∧ (n < 2)) ⇒ (vect.Sum[m][n] ≡ vect.Const[m][n]+vect.Const1[m][n]);	store	Preservation	✗		✗	✓	✗
sum_matrix.c	f		loop variant 2-i;	store	Decreasing	✓		✓	✓	✓
sum_matrix.c	f		loop variant 2-i;	store	Positive	✓		✓	✓	✓
sum_matrix.c	f		loop variant 2-j;	store	Decreasing	✓		✓	✓	✓
sum_matrix.c	f		loop variant 2-j;	store	Positive	✓		✓	✓	✓
sum_matrix.c	f		ensures ∀ ℤ m, ℤ n; ((0 ≤ m) ∧ (m < 2)) ∧ ((0 ≤ n) ∧ (n < 2)) ⇒ (vect.Sum[m][n] ≡ vect.Const[m][n]+vect.Const1[m][n])	store	Proof Obligation	?		✓	✓	✓

FIGURE 4 – Resultat de la vérification

6 Conclusion

Pour répondre à l'évolution de la complexité des applications critiques et aux difficultés de leur vérification, les approches formelles sont de plus en plus souvent exploitées pour des systèmes industriels réels et commencent à être intégrées dans les standards.

GENEAUTO fait partie de nombreux projets qui s'attachent à appliquer et intégrer les techniques de vérification formelle aux systèmes critiques. Son objectif principal est le développement d'un générateur de code automatique qui puisse être qualifié à moyen terme pour la certification des systèmes critiques développés avec ce générateur. Certaines techniques de vérification et de validation formelle s'avèrent plus prometteuses que d'autres au regard de leur intégration possible dans des projets réels et de leur passage à l'échelle.

L'approche de vérification présentée ici dont les premiers résultats expérimentaux semblent présager des pistes intéressantes va nécessiter dans l'avenir le développement d'un outillage spécifique afin de faciliter son appréhension et son utilisation par l'utilisateur final. La définition d'un métamodèle et de contraintes OCL servant de base à cette représentation permettra la création de l'outillage adapté et la définition de la transformation. Il conviendra aussi d'intégrer d'autres aspects de vérification que ceux présentés ici afin de compléter la vérification de ce générateur de code.

Références

[1] Bruno Barras and Bruno Bernardo. The implicit calculus of constructions as a programming language with dependent types. In Roberto M. Amadio, editor, *FoSSaCS*, volume 4962 of *Lecture Notes in Computer Science*, pages 365–379. Springer, 2008.

[2] Sandrine Blazy and Xavier Leroy. Mechanized semantics for the clight subset of the c language. *J. Autom. Reasoning*, 43(3) :263–288, 2009.

[3] Jan Olaf Blech, Sabine Glesner, Johannes Leitner, and Steffen Mülling. Optimizing code generation from ssa form : A comparison between two formal correctness proofs in isabelle/hol. In *Proceedings of the COCV-Workshop (Compiler Optimization meets Compiler Verification), 8th European Conferences on Theory and Practice of Software (ETAPS 2005)*, pages 33–51. Elsevier, April 2005.

[4] David Cachera, Thomas P. Jensen, David Pichardie, and Vlad Rusu. Extracting a data flow analyser in constructive logic. *Theor. Comput. Sci.*, 342(1) :56–78, 2005.

[5] Maulik A. Dave. Compiler verification : a bibliography. *ACM SIGSOFT Software Engineering Notes*, 28(6) :2, 2003.

[6] François-Xavier Dormoy. Scade 6 : A model based solution for safety critical software development. In *European Congress Embedded Real-Time Software (ERTS)*, 2008.

[7] Stéphane Glondu. Extraction certifiée dans coq-en-coq. In *Journées Francophones des Langages Applicatifs (JFLA)*, Saint-Quentin sur Isère, France, January 2009.

[8] Nicolas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. The synchronous dataflow programming language lustre. *Proceedings of the IEEE*, 79(9) :1305–1320, September 1991.

[9] Nassima Izerrouken, Marc Pantel, and Xavier Thirioux. Machine-checked sequencer for critical embedded code generator. In Karin Breitman and Ana Cavalcanti, editors, *ICFEM*, volume 5885 of *Lecture Notes in Computer Science*, pages 521–540. Springer, 2009.

[10] Nassima Izerrouken, Marc Pantel, Xavier Thirioux, and Olivier Ssi Yan Kai. Integrated formal approach for qualified critical embedded code generator. In María Alpuente, Byron Cook, and Christophe Joubert, editors, *FMICS*, volume 5825 of *Lecture Notes in Computer Science*, pages 199–201. Springer, 2009.

[11] Nassima Izerrouken, Xavier Thirioux, Marc Pantel, and Martin Strecker. Certifying an automated code generator using formal tools : Preliminary experiments in the geneauto project. In *European Congress on Embedded Real-Time Software (ERTS)*. SIA, 2008.

[12] Gérard Ladier. Opees workshop on the use of formal technologies for the v & v of code generators. Informal meeting, sept 2007.

[13] Thierry Lecomte. Safe and reliable metro platform screen doors control/command systems. In Jorge Cuéllar, T. S. E. Maibaum, and Kaisa Sere, editors, *FM*, volume 5014 of *Lecture Notes in Computer Science*, pages 430–434. Springer, 2008.

[14] Xavier Leroy. Formal certification of a compiler back-end or : programming a compiler with a proof assistant. In J. Gregory Morrisett and Simon L. Peyton Jones, editors, *POPL*, pages 42–54. ACM, 2006.

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

[16] Xavier Leroy. Verified squared : does critical software deserve verified tools ? In *38th symposium Principles of Programming Languages*, pages 1–2. ACM Press, 2011. Abstract of invited lecture.

[17] Pierre Letouzey. Extraction in coq : An overview. In Arnold Beckmann, Costas Dimitracopoulos, and Benedikt Löwe, editors, *CiE*, volume 5028 of *Lecture Notes in Computer Science*, pages 359–369. Springer, 2008.

[18] Robin Milner and R. Weyhrauch. Proving compiler correctness in a mechanised logic. *Machine Intelligence*, (7), 1972.

[19] George C. Necula and Peter Lee. Research on proof-carrying code for untrusted-code security. In *IEEE Symposium on Security and Privacy*, page 204. IEEE Computer Society, 1997.

[20] Bruno Pagano, Olivier Andrieu, Benjamin Canou, Emmanuel Chailloux, Jean Louis Colaco, Thomas Moniot, and Philippe Wang. Certified development tools implementation in objective caml. In *PADL*, pages 2–17, Toulouse, France, 2008.

[21] Bruno Pagano, Olivier Andrieu, Thomas Moniot, Benjamin Canou, Emmanuel Chailloux, Philippe Wang, Pascal Manoury, and Jean-Louis Colaço. Experience report : using objective caml to develop safety-critical embedded tools in a certification framework. In Graham Hutton and Andrew P. Tolmach, editors, *ICFP*, pages 215–220. ACM, 2009.

[22] Amir Pnueli, Michael Siegel, and Eli Singerman. Translation validation. In Bernhard Steffen, editor, *TACAS*, volume 1384 of *Lecture Notes in Computer Science*, pages 151–166. Springer, 1998.

[23] Famantanantsoa Randimbivololona, Jean Souyris, Patrick Baudin, Anne Pacalet, Jacques Raguideau, and Dominique Schoen. Applying formal proof techniques to avionics software : A pragmatic approach. In Jeannette M. Wing, Jim Woodcock, and Jim Davies, editors, *World Congress on Formal Methods*, volume 1709 of *Lecture Notes in Computer Science*, pages 1798–1815. Springer, 1999.

[24] Xavier Rival. Symbolic transfer function-based approaches to certified compilation. In Neil D. Jones and Xavier Leroy, editors, *POPL*, pages 1–13. ACM, 2004.

[25] Jean Souyris and David Delmas. Experimental assessment of astrée on safety-critical avionics software. In Francesca Saglietti and Norbert Oster, editors, *SAFECOMP*, volume 4680 of *Lecture Notes in Computer Science*, pages 479–490. Springer, 2007.

[26] Stephan Thesing, Jean Souyris, Reinhold Heckmann, Famantanantsoa Randimbivololona, Marc Langenbach, Reinhard Wilhelm, and Christian Ferdinand. An abstract interpretation-based timing validation of hard real-time avionics software. In *DSN*, pages 625–. IEEE Computer Society, 2003.

[27] Andres Tooms, Nassima Izerrouken, Tonu Naks, Marc Pantel, and Olivier Ssi-Yan-Kai. Towards reliable code generation with an open tool : Evolutions of the gene-auto toolset. In *European symposium on Real Time Software and Systems (ERTS^2)*. SIA, 2010.

[28] Andres Tooms, Tonu Naks, Marc Pantel, Marcel Gandriau, and Indra Wati. Gene-auto - an automatic code generator for a safe subset of simulink-stateflow and scicos. In *European symposium on Real Time Systems (ERTS)*. SIA, 2008.

[29] Jean-Baptiste Tristan and Xavier Leroy. Formal verification of translation validators : a case study on instruction scheduling optimizations. In George C. Necula and Philip Wadler, editors, *POPL*, pages 17–27. ACM, 2008.

[30] Kumar Neeraj Verma, Jean Goubault-Larrecq, Sanjiva Prasad, and S. Arun-Kumar. Reflecting bdds in coq. In Jifeng He and Masahiko Sato, editors, *ASIAN*, volume 1961 of *Lecture Notes in Computer Science*, pages 162–181. Springer, 2000.

[31] Anna Zaks and Amir Pnueli. Program analysis for compiler validation. In Shriram Krishnamurthi and Michal Young, editors, *PASTE*, pages 1–7. ACM, 2008.