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

Arnaud Dieumegard
Arnaud.Dieumegard@enseeiht.fr

Marc Pantel
Marc.Pantel@enseeiht.fr

**Résumé**

Dans cette communication, nous presentons une strategie de verification d'un générateur automatique de code pour systèmes critiques embarqués (GENEAUTO). Il s'agit pour nous de définir une méthodologie pouvant être exploitée par les experts métiers utilisateurs du générateur afin qu'ils puissent définir eux-mêmes les blocs utilisés et leur semantique. La vérification du code généré doit s'appuyer sur des techniques abordables pour les futurs utilisateurs. Pour cela, ils doivent fournir une spécification formelle de chacun de leurs blocs métiers qui sera ensuite utilisée pour générer des annotations sur le code 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

**Résumé**

In this communication, we present the GENEAUTO code generator verification strategy for the traduction of SIMULINK systems to C code for critical systems development. The definition of a verification method that can be used by domain experts is the entry point of the method. They can define themselves specific blocks and their semantics in order to verify the code generation. The generated code verification must rely on high level techniques in order to be usable by the targeted users. The specified semantics is used to generate ACSL annotations verified with SMT solvers in the FRAMA-C toolset.

**Keywords :** Critical systems, verification, code generation, ACSL, SMT solver

## 1 Introduction

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). Afin de satisfaire aux normes de certification, les approches classiques pour la qualification des générateurs de code reposent principalement sur le respect d'un processus de développement très détaillé.

L'utilisation de techniques classiques (peu efficaces et très couteuses) pour la vérification du code source conduit à l'utilisation de méthodes formelles pour effectuer une vérification exhaustive. 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[8, 5]. La version C du standard aéronautique DO178/ED12 contient désormais 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.

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

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

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. Un point important dans le projet était l'expérimentation de méthodes formelles [4, 3, 2] pour la vérification de l'implémentation des composants et les échanges avec les autorités de certification.

Dans la section suivante, nous présenterons brièvement nos choix en matière de techniques de vérification. Dans la section 3 nous présentons les outils que nous utilisons pour notre étude. Dans la section 4 nous entrons plus en détail dans notre expérimentation, nous concluons et traçons des perspectives dans la section 5.

## 2 Sélection d'une technique de vérification formelle

Lors de la première phase de GENEAUTO, nous avons étudié l'approche formelle la mieux adaptée au développement d'un générateur qui devra être qualifié. Deux critères se sont révélés décisifs dans la sélection d'une méthode de vérification : la réduction du risque d'anomalies résiduelles et la durée de la phase de vérification. 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. De plus, lors de la détection d'une anomalie sur le système en service, celle-ci doit être corrigée et déployée rapidement pour réduire les coûts d'immobilisation et les risques si le service n'est pas suspendu.

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 [2]. Par contre, lorsque les utilisateurs finaux doivent pouvoir intervenir dans la spécification et la vérification, la complexité 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 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 plus accessibles pour l'utilisateur en cas d'échec dans la vérification et disposant d'outils de vérification matures et qualifiables à moyen terme.

## 3 Outils utilisés pour notre approche

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 [3] 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.

**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. Un modèle SIMULINK est représenté graphiquement par des diagrammes composés de blocs reliés par 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. Plusieurs blocs ont été choisis pour notre étude en fonction de leur intérêt. Nous ne décrirons dans la suite que le bloc SIMULINK Sum.

3. ANSI/ISO C Specification Language

**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 [4] et de l'analyseur de code source CADUCEUS [5] du projet WHY [6]. Les annotations ACSL se présentent sous la forme de commentaires du code C dont les caractères de début (// ou /*) sont post fixés 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). Pour plus d'informations, voir le document de référence [7].

**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. Il permet d'effectuer une vérification déductive du code généré par l'outil GENEAUTO par le biais du greffon WP (Weakest Preconditions).

**Le greffon WP** L'extension WP implante un calcul de plus faible pré conditions 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.

Grâce 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.

# 4 Intégration des approches formelles dans GENEAUTO

L'intégration d'approches formelles dans un processus industriel pour le développement d'outils qualifiés est une des innovations de GENEAUTO. Il s'agit de prendre en compte des contraintes industrielles (exigences en langage naturel, langages de modélisation utilisés, 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 telles que des approches s'appuyant sur une définition de la sémantique des langages source et cible et sur la preuve de préservation de cette sémantique ou d'autres approches basées sur l'observation des entrées/sorties [1] ou encore les modifications de l'environnement (mémoire)[6, 7].

**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.

4. http://www.cs.ucf.edu/~leavens/JML/
5. http://caduceus.lri.fr/
6. http://why.lri.fr/
7. http://frama-c.com/download/acsl_1.5.pdf

**Entrées** Le nombre d'éléments en entrée est illimité mais les entrées doivent toutes être de même type de données. Si les types de signaux sont différents, alors les seules combinaisons de types de signaux possibles sont : Scalaire-Vecteur et Scalaire-Matrice. Si une seule entrée est donnée alors elle doit être de type vecteur ou scalaire.
**Sorties** Une seule sortie, son type de signal et de données est le même que celui des entrées.
– Si les entrées sont des scalaires ou un seul vecteur : la sortie sera un scalaire égal à la somme des scalaires d'entrée ou la somme des composantes du vecteur d'entrée.
– Si les entrées sont des vecteurs (potentiellement des scalaires) : la sortie sera un vecteur somme des vecteurs d'entrée (composante par composante).
– Si les entrées sont des matrices (potentiellement des scalaires) : la sortie sera une matrice somme des matrices d'entrée (composante par composante).

FIGURE 1 – Specification en langage naturel

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.

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

**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 : Figure 2 pour le bloc Sum).

Le type de structures de données choisi comme entrée du bloc lors de la conception du système étant variable (scalaires, vecteurs, matrices), il a 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. Un exemple de code annoté correspondant au bloc Sum pour lequel les entrées sont des vecteurs peut être trouvé au listing 1 (Nous avons choisi dans ce cas d'avoir l'entrée $i5$ de type scalaire). Il est nécessaire d'ajouter à la génération d'annotations pour les blocs, une génération d'annotations pour les signaux. Ces annotations dépendent uniquement du type du signal et de sa dimension, le comportement attendu du code généré est toujours la recopie des valeurs d'entrée sur la cible su signal (voir listing 2).

**Correspondance annotations/spécification 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, il s'agit ici d'annotations utilisant la quantification universelle *forall* présente dans les annotations *loop invariant*. 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 (*ensure*) et les pré-conditions (*requires*) sont spécifiées au début du bloc.

– $\mathcal{I}$ l'ensemble des signaux d'entrée de type $\mathbb{T}$
– $n \in \mathbb{N}$ le nombre de signaux d'entrée
– $d, e \in \mathbb{N}$ la dimension des signaux d'entrée (seulement $d$ est nécessaire si les entrées sont des vecteurs, $d$ et $e$ sont nécessaires en cas d'utilisation de matrices).

L'entrée doit être d'une des formes suivantes :

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

Selon les formalisations des entrées, voici les formalisations des sorties ($S$) :

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

Concernant les points 3 et 4, il est nécessaire que :
$\forall \triangle \in \{+,-\}, \forall i \in \{1,d\} \wedge \forall j \in \{1,e\}$, $\{s, a_{i,j}\} \in \mathbb{T}$,
$$s \triangle X = \begin{pmatrix} s \triangle a_{1,1} & \cdots & s \triangle a_{1,j} \\ \vdots & \ddots & \vdots \\ s \triangle a_{i,1} & \cdots & s \triangle a_{i,j} \end{pmatrix}$$

FIGURE 2 – Formalisation de la spécification du bloc Sum

– Les annotations spécifiques à l'algorithme généré par GENEAUTO. Ce sont les autres annotations des exemples du listing 1. Elles permettent de s'assurer de la terminaison des boucles (loop variant).

```
/*@ requires \forall integer m;
      0 <= m < 4 ==> \valid(&S+m) && \valid(&x1+m) && \valid(&x2+m);
    ensures \forall integer m; 0 <= m < 4 ==> S[m] == x1[m] + x2[m];*/
{
/*@ loop invariant 0 <= index <= 4;
    loop invariant \forall integer m; index <= m < 4 ==> S[m] == \at(S[m],Pre);
    loop invariant \forall integer m; 0 <= m < index ==> S[m] == x1[m] + x2[m];
    loop variant 4 - index;*/
for (int index = 0; index < 4; index++){
  S[index] = x1[index] + x2[index];}}
```

Listing 1 – Code et annotations générés pour le bloc Sum (entrée du bloc est un vecteur de taille 4)

```
/*@  ensures \forall integer n; 0 <= n < 4 ==> b2.i1[n] == b1.o1[n];*/
{
/*@  loop invariant 0 <= index <= 4;
     loop invariant \forall integer n; 0 <= n < index ==> b2.i1[n] == b1.o1[n];
     loop variant 4-index;*/
for(int index=0; index < 4; ++index){
   b2.i1[index] = b1.o1[index];}}
```

Listing 2 – Code type et annotations générés pour un signal (vectoriel)

**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 3 le

résultat de la vérification à l'aide su solveur SMT Simplify. Le détail des annotations est donné et, pour chacune, le resultat du solveur est affiché : [ Valid ]. Il est possible de combiner plusieurs solveurs SMT afin d'obtenir une analyse plus performante dans des cas plus compliqués.

# 5 Perspectives et Conclusion

GENEAUTO fait partie de nombreux projets qui s'attachent à appliquer et intégrer les techniques de vérification formelle aux systèmes critiques. 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.

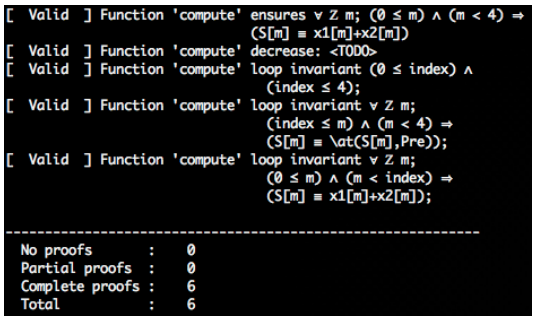


FIGURE 3 – Resultat de la vérification

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 vers les annotations. L'approche de génération d'annotations sur le code devra être généralisée pour permettre l'extensibilité. Le passage à l'echelle de l'aproche devra aussi être étudié. 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] S. Blazy and X. Leroy. Mechanized semantics for the clight subset of the c language. *J. Autom. Reasoning*, 43(3) :263–288, 2009.

[2] N. Izerrouken, M. Pantel, and X. Thirioux. Machine-checked sequencer for critical embedded code generator. In *ICFEM*, volume 5885, 09.

[3] N. Izerrouken, M. Pantel, X. Thirioux, and O. S. Y. Kai. Integrated formal approach for qualified critical embedded code generator. In *FMICS*, volume 5825, 09.

[4] N. Izerrouken, X. Thirioux, M. Pantel, and M. Strecker. Certifying an automated code generator using formal tools : Preliminary experiments in the geneauto project. In *ERTS*, 08.

[5] T. Lecomte. Safe and reliable metro platform screen doors control/command systems. In *FM*, volume 5014, 2008.

[6] A. Pnueli, M. Siegel, and E. Singerman. Translation validation. In B. Steffen, editor, *TACAS*, volume 1384 of *Lecture Notes in Computer Science*, pages 151–166. Springer, 1998.

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

[8] J. Souyris and D. Delmas. Experimental assessment of astrée on safety-critical avionics software. In *SAFECOMP*, volume 4680, 07.