

Remerciements

Les remerciements

Résumé

Votre résumé ici (1 à 2 pages).

Table des matières

Remerciements	1
Résumé	3
Glossaire	7
Introduction	9
1 Présentation de la vérification formelle	11
2 Présentation de la plateforme Frama-C	13
2.1 Caractéristiques générales de la plateforme Frama-C	13
2.1.1 Architecture de Frama-C	13
2.1.2 Interface graphique	14
2.2 ACSL	16
2.2.1 Présentation	16
2.2.2 Utilisations d'ACSL avec Frama-C	17
2.3 EVA	17
2.3.1 Présentation et objectifs	17
2.3.2 Gestion des boucles	17
2.3.3 Utilisation de l'interface graphique	18
2.4 WP	19
2.4.1 Présentation et objectifs	19
2.4.2 Logique de Hoare et plus faible précondition	19
2.4.3 Contrats de fonction	20
2.4.4 Les boucles	22
2.4.5 Utilisation de proveurs externes avec WP	23
2.4.6 Utilisation de l'interface graphique	23
2.5 Stratégie d'utilisation de Frama-C	27
3 Projet analysé	29
3.1 Présentation de WooKey	29
3.2 Présentation de la bibliothèque USBctrl	30
3.2.1 Généralités sur le protocole USB et spécifications de l'USB 2.0	30
3.2.2 Rôle de la bibliothèque USBctrl dans le module USB de WooKey	30
3.3 Code source de la bibliothèque USBctrl	34
3.3.1 Description	34
3.3.2 Rappel sur le langage C	34
4 Utilisation de FramaC sur la bibliothèque USBctrl	37
4.1 Prise en main de Frama-C	37
4.1.1 Installation	37
4.1.2 Tutorial	37
4.2 Utilisation de Frama-C sur la bibliothèque USBctrl	37
4.2.1 Parsing du code source de la bibliothèque USBctrl de WooKey	38

4.2.2	Utilisation d'EVA	38
4.2.3	Utilisation de WP	44
5	Résultats obtenus	47
5.1	Résultats de l'analyse réalisée avec EVA	47
5.1.1	Travail réalisé	47
5.1.2	RTE découverts	48
5.1.3	Problèmes rencontrés	48
5.1.4	Pistes d'amélioration	48
5.2	Résultats obtenus avec WP	48
5.2.1	Travail réalisé	48
5.2.2	Problèmes rencontrés	48
5.2.3	Résultats obtenus	49
5.3	Retour d'expérience sur l'utilisation de Frama-C	49
	Conclusion	51
A	Point d'entrée dans la bibliothèque USBctrl	53

Glossaire

Introduction

produit de sécurité
programmes critiques
enjeux de sécurité
complexité des programmes - criticité de certains programmes développement de l'embarqué nécessite
d'être sûr que les programmes sont sécurisés Absences de bug à l'exécution
IMPORTANT : avoir confiance dans le logiciel / le matériel
software / hardware engineering
safety critical / security critical
quelques mots d'intro sur wookey
Dans ce stage, j'ai utilisé la plateforme FRAMA-C afin de vérifier formellement une bibliothèque de
wookey en charge notamment de la gestion de l'USB 2.0.

Chapitre 1: Présentation de la vérification formelle

La vérification formelle consiste à raisonner rigoureusement, à l'aide de logique mathématique, sur des programmes informatiques afin de démontrer leur validité par rapport à une certaine propriété. Un exemple de propriété est l'absence de Run Time Error¹ (RTE dans le reste du document)

Cette vérification est basée sur la sémantique des programmes, c'est-à-dire sur le lien entre le langage signifiant (le langage de programmation) et le langage signifié (description mathématique du programme sous forme logique, d'automates ou autre). Les mathématiques sont donc utilisées pour concevoir, réaliser ou vérifier un système informatique (spécification/conception/vérification formelle). La vérification formelle présente l'avantage d'être précise et non ambiguë (toutes les propriétés initialement exprimées en langage naturel sont décrites de manière mathématique). Ainsi, l'objectif de vérifier formellement un programme (ou de le prouver) est de s'assurer que, quelle que soit l'entrée fournie au programme, si elle respecte la spécification, alors le programme fera ce qui est attendu. Les mathématiques pourront prouver que le programme ne peut avoir que les comportements qui sont spécifiés et que les erreurs d'exécution n'en font pas partie.

La vérification formelle peut être illustrée en raisonnant sur l'identité remarquable suivante : $(a + b)^2 = a^2 + b^2 + 2ab$

Il est possible de tester cette équation avec plusieurs valeurs pour vérifier l'égalité entre les deux termes, mais cette vérification ne peut pas être exhaustive ou cela nécessiterait de tester l'équation sur $2^{31} * 2^{31}$ valeurs. Il est également possible de prouver ce résultat simplement en développant et en factorisant pour montrer l'équivalence. Il s'agit dans ce cas d'une vérification formelle.

De manière générale, la vérification formelle comprend plusieurs étapes :

- la spécification² formelle du programme informatique en utilisant les mathématiques, qui consiste au passage d'un langage informatique à un langage mathématique ;
- la vérification du respect de certaines propriétés de la spécification : propriétés fonctionnelles, de sûreté et de sécurité, par exemple l'absence de RTE.

Deux notions importantes permettent par ailleurs de caractériser la vérification formelle :

- La complétude : la vérification formelle ne comporte pas de faux positifs, chaque erreur détectée est une véritable erreur (sous approximation du code analysé). Cela ne permet toutefois pas de conclure à l'absence d'erreur dans le code ;
- La correction : la vérification formelle ne comporte pas de faux négatifs : toutes les erreurs potentielles sont détectées (sur approximation du code analysé). Néanmoins, des faux positifs peuvent être relevés lors de la vérification formelle, ce qui nécessite un travail d'analyse supplémentaire.

1. Une RTE correspond à une erreur survenue lors de l'exécution d'un programme, pouvant provoquer son arrêt non prévu mais aussi pouvant mener à un comportement indéfini du programme

2. Une spécification est une description d'exigences à satisfaire par un matériel, un produit ou un service

Chapitre 2: Présentation de la plateforme Framac

2.1 Caractéristiques générales de la plateforme Framac

La vérification formelle, présentée dans le chapitre précédent, peut être réalisée à l'aide de plusieurs outils qui dépendent du langage de programmation à analyser. Dans le cadre de ce stage, le langage de programmation à analyser étant le C, la plateforme Framac (Framework for modular analysis of C programs) a été utilisée.

Framac est une plate-forme open-source, modulaire et collaborative dédiée à l'analyse du langage C. Framac rassemble plusieurs techniques d'analyse dans un seul cadre, sous forme de greffons (l'architecture de Framac est présentée dans le chapitre 2.1.1). Framac est développé par le Laboratoire de sûreté des logiciels du CEA-LIST ainsi que par l'équipe de recherche Toccata de l'INRIA.

Les analyses réalisées avec Framac peuvent être statiques (sans exécution de code) ou dynamiques (avec exécution de code). Au cours de ce stage, seules des analyses statiques formelles ont été réalisées. Par ailleurs, il est important de noter que pour ces analyses, l'objectif de Framac est de fournir des garanties quant à l'absence d'erreur lors de l'exécution du code¹ en prévenant l'utilisateur à chaque fois qu'une erreur est susceptible de se produire², ainsi qu'à l'absence de déviation par rapport aux spécifications fonctionnelles du code.

2.1.1 Architecture de Framac

Comme précisé en début de chapitre, Framac est modulaire. Cela signifie que les différents modules, aussi appelés greffons, sont connectés à une plateforme centrale, le noyau. Chaque greffon permet d'analyser et /ou d'annoter le code source, analyses et annotations qui peuvent être utilisées par d'autres greffons dans d'autres analyses.

Framac est installé avec des greffons de base, open-source, développés et maintenus par le CEA-LIST. Néanmoins, des greffons supplémentaires peuvent être fournis, par le CEA-LIST ou par des développeurs externes au projet Framac (par exemple par des industriels), et installés séparément du noyau. Cela signifie que Framac n'est pas limité au jeu d'analyses initialement installé. Les greffons peuvent par ailleurs collaborer entre eux, en se servant des analyses réalisées par d'autres greffons.

La figure suivante représente les greffons open-source développés par le CEA-LIST, et illustre la complexité et la puissance de la plateforme Framac.

1. dans le reste de ce rapport, ces erreurs seront nommées RTE, pour run time errors
2. ce qui peut conduire à l'émission de faux positifs que l'utilisateur doit analyser

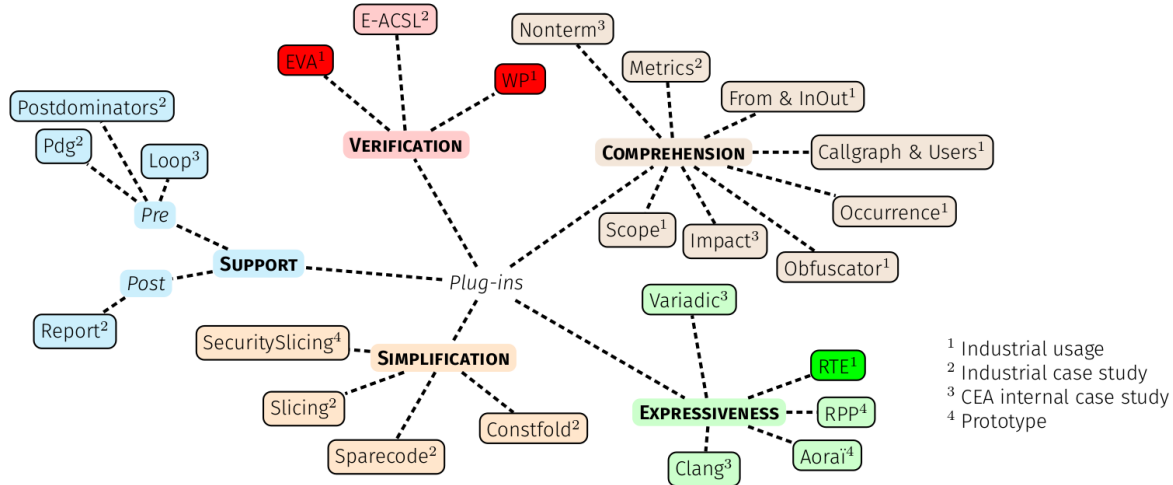


FIGURE 2.1 – Greffons open-source du CEA

Les greffons utilisés dans le cadre de ce stage sont EVA (Evolved Value Analysis) et WP (Weakest Precondition). Ces deux greffons sont présentés dans des chapitres ci-dessous (respectivement les chapitres 2.3 et 2.4). Néanmoins, avant d'expliquer leur fonctionnement et leur utilité, il est nécessaire de présenter le langage de spécification utilisé par Frama-C et ses greffons.

2.1.2 Interface graphique

Frama-C peut s'utiliser en ligne de commandes ou à l'aide d'une interface graphique. Il est possible d'utiliser cette interface graphique, installée par défaut, afin d'effectuer des analyses avec Frama-C ou afin de visualiser le résultat des analyses réalisées en ligne de commandes. Néanmoins, l'utilisation de Frama-C en ligne de commandes présente l'avantage de permettre l'utilisation des nombreuses options de Frama-C et de ses greffons. C'est pourquoi, dans ce stage, Frama-C a été utilisé en ligne de commandes pour effectuer les analyses, l'interface graphique permettant de visualiser les résultats obtenus.

A travers l'interface graphique, il est possible de :

- naviguer dans les différents fichiers composant le code source à analyser, fonction par fonction ;
- visualiser, le cas échéant, les annotations automatiques ajoutées lors de l'analyse du programme informatique ;
- visualiser les résultats de la vérification formelle ;
- examiner les avertissements et les alarmes générés lors de l'analyse ;
- relancer l'analyse après la modification du code source, les options de Frama-C définies lors de la première analyse étant conservées.

La figure suivante présente un exemple d'utilisation de l'interface graphique sur le code source analysé lors de ce stage.

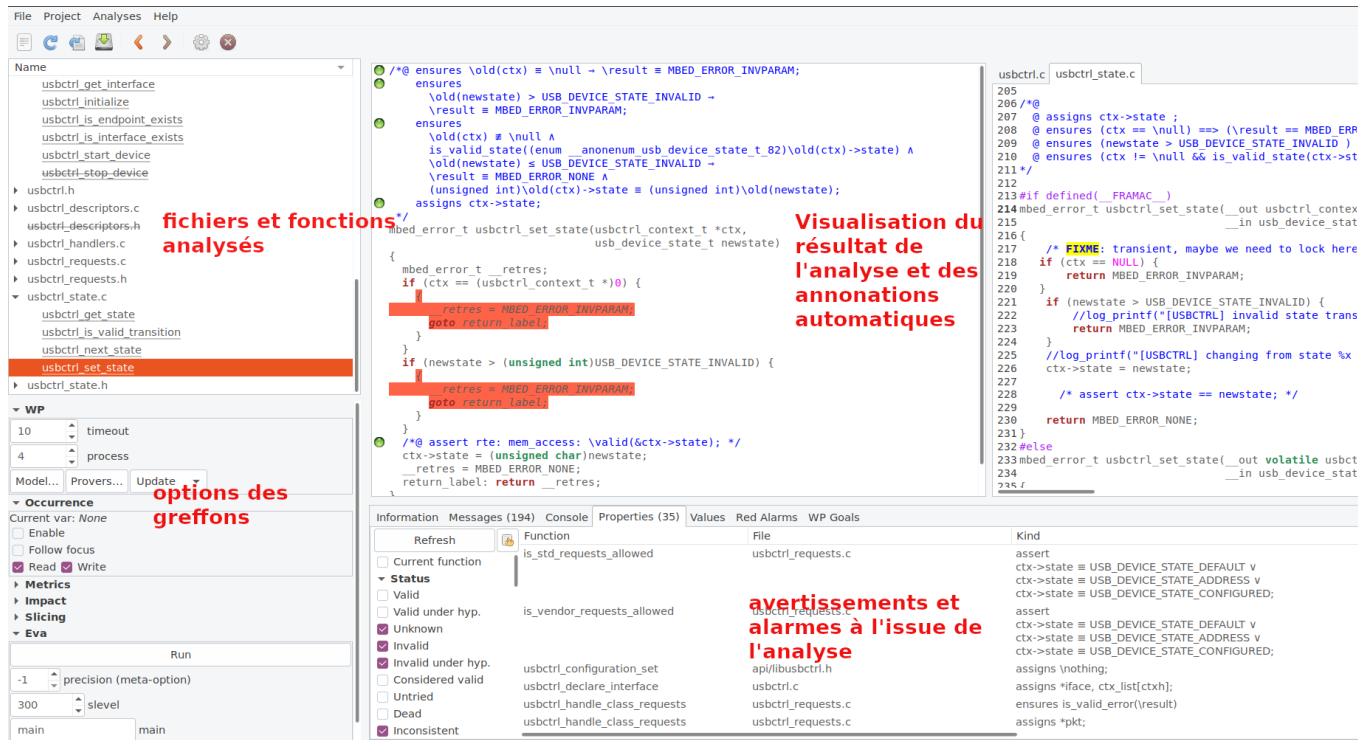
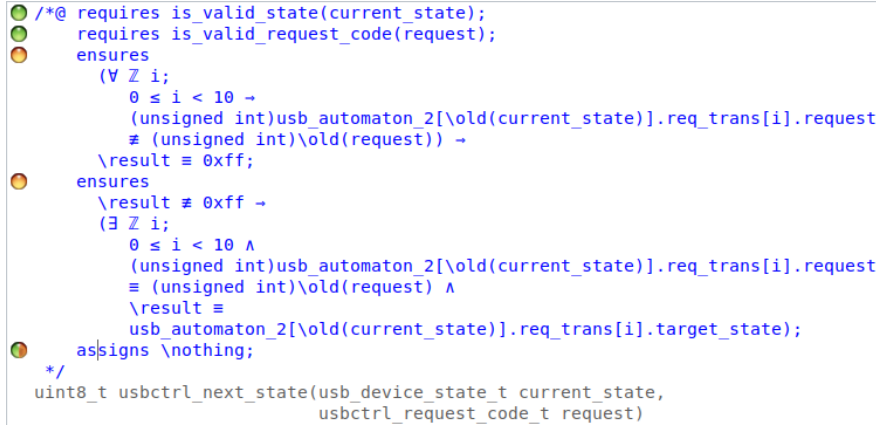


FIGURE 2.2 – Interface graphique de Frama-C

La visualisation des résultats à l'aide de l'interface graphique est très claire :

- un point vert signifie que la propriété est valide : "surely valid : verified (including all of its dependencies)";
- un point orange signifie que Frama-C n'est pas capable de déterminer si la propriété est valide ou fautive : "Unknown : a verification has been attempted, but without conclusion". Il est important de noter que dans ce cas, cette hypothèse sera considérée comme valide dans la suite de l'analyse, ce qui amène au point suivant ;
- un point vert et orange signifie que la propriété est valide, mais qu'elle dépend néanmoins de propriétés qui n'ont pas pu être prouvées : "valid under hypotheses : verified (but has dependencies with unknown status)". Cela signifie que la propriété n'est en réalité pas validée, car la non vérification de certaines propriétés ne permet pas d'avoir confiance dans la propriété principale ;
- un point rouge signifie que la propriété est invalide : "invalid under hypotheses : refuted".



```

/*@ requires is_valid_state(current_state);
   requires is_valid_request_code(request);
   ensures
     (\forall \mathbb{Z} i;
      0 \leq i < 10 \rightarrow
      (unsigned int)usb_automaton_2[\old(current_state)].req_trans[i].request
      \neq (unsigned int)\old(request)) \rightarrow
     \result \equiv 0xff;
   ensures
     \result \neq 0xff \rightarrow
     (\exists \mathbb{Z} i;
      0 \leq i < 10 \wedge
      (unsigned int)usb_automaton_2[\old(current_state)].req_trans[i].request
      \equiv (unsigned int)\old(request) \wedge
      \result \equiv
      usb_automaton_2[\old(current_state)].req_trans[i].target_state);
   assigns \nothing;
*/
uint8_t usbctrl_next_state(usb_device_state_t current_state,
                          usbctrl_request_code_t request)

```

FIGURE 2.3 – Visualisation des résultats de preuve avec l'interface graphique

2.2 ACSL

2.2.1 Présentation

Comme expliqué dans les paragraphes précédents, la vérification formelle peut nécessiter de spécifier le code source, c'est à dire de décrire le comportement attendu du code afin de vérifier que le code respecte bien ce comportement.

Dans le cadre de Frama-C, ces spécifications doivent s'écrire dans un langage particulier, dont la syntaxe est proche de celle du langage de programmation C : le langage ACSL (ANSI/ISO C Specification Language).

Les spécifications écrites en ACSL peuvent être automatiquement manipulées par des outils de vérification formelle, de la même manière qu'un langage de programmation est manipulé par un compilateur, par opposition à des commentaires informels qui ne peuvent être utiles qu'aux humains. De plus, le langage ACSL est compris par les différents greffons de Frama-C.

Le langage ACSL est très souple et permet d'écrire des propriétés élémentaires, par exemple "cette fonction nécessite un pointeur valide en entrée", aussi bien que des propriétés plus complexes, par exemple "cette fonction nécessite une liste chaînée non vide d'entiers en entrée, et retourne l'entier le plus grand en sortie".

Le langage ACSL présente les caractéristiques principales suivantes :

- utilisation d'expressions écrites en C. En particulier, la syntaxe des expressions booléennes est identiques à celle du C ;
- annotations délimitées par `/*@` pour une annotation sur une ligne, ou `/* @ */` pour un bloc d'annotations (annotations semblables aux commentaires en C, avec l'ajout du signe `@` pour introduire des spécifications ACSL) ;
- utilisation du typage du langage C (int, float, double etc.), ainsi qu'utilisation de types ACSL plus généraux (types `Z` et `R`, pour les entiers et les réels) ;
- existence de prédicats³ et de fonctions logiques⁴ pré-implémentés pour exprimer des propriétés souvent utiles en C. Par exemple, pour manipuler les pointeurs, il est possible d'utiliser avec ACSL les prédicats suivants :

- `\valid(p)` : pointeur `p` non null, accessible en lecture et en écriture ;
- `\valid_read(p)` : pointeur `p` non null, accessible en lecture seulement ;
- `\separated(p,q)` : les pointeurs `p` et `q` ne partagent pas la même adresse mémoire.

3. Un prédicat est une propriété pouvant être vraie ou fausse

4. Les fonctions logiques permettent de décrire des fonctions mathématiques qui, contrairement aux prédicats, permettent de renvoyer différents types

2.2.2 Utilisations d'ACSL avec Frama-C

Les principales utilisations d'ACSL avec Frama-C sont les suivantes :

- annotations automatiques dans le code source analysé lors de l'utilisation de certains greffons. Ces annotations sont visibles lors de l'utilisation de l'interface graphique, le code source n'étant pas modifié. Un exemple d'ajout d'annotations est le suivant :

```
/*@ assert rte: mem_access: \valid(a); */  
*a = (unsigned char)1;
```

Cette annotation signifie que l'adresse mémoire du pointeur "a" doit être valide avant son assignation.⁵

- ajout d'annotations à des fins de débogage ou pour permettre à certains greffons de vérifier le code analysé, en utilisant par exemple des assertions ;
- définitions de contrats de fonction afin de spécifier le comportement de la fonction qui devra être vérifié par Frama-C. Les contrats de fonction, utilisés durant ce stage avec WP, sont décrits dans le paragraphe 2.4.3.

2.3 EVA

2.3.1 Présentation et objectifs

EVA (Evolved Value Analysis) est le premier greffon de Frama-C utilisé au cours de ce stage. EVA a pour objectif d'analyser automatiquement le code, à l'aide d'une analyse des valeurs par interprétation abstraite, afin d'estimer les valeurs possibles des différentes variables présentes dans le code source et de détecter les RTE potentielles (accès mémoire invalide, variables non initialisées, division par 0...) EVA réalisant une analyse statique, les valeurs possibles des variables sont déterminés sans exécuter le code en réalisant une sur-approximation du code source. Des alarmes et avertissements sont par ailleurs émis pour chaque opération invalide présente dans le code ainsi que pour les annotations ACSL invalides. La sur-approximation du code source permet au greffon de prouver avec un haut degré de confiance l'absence d'erreurs lors de l'exécution du code en garantissant l'absence de faux-négatifs (EVA est correct, dans le sens défini au chapitre précédent). Enfin, l'utilisation d'EVA ne se limite pas à la seule détection des RTE. En effet, EVA permet aussi de trouver le cas échéant des portions de code mort qui peuvent être dues à des erreurs de programmation (par exemple, une condition jamais atteinte à cause d'un test erroné quelques lignes plus tôt dans le code).

L'utilisation d'EVA nécessite un point d'entrée dans le code pour commencer l'analyse du code. Cela signifie que le code analysé nécessite une fonction d'entrée (par exemple une fonction main, mais ce n'est pas obligatoire) à partir de laquelle les différentes fonctions à analyser seront appelées et dans laquelle les différentes variables du programme seront initialisées.

2.3.2 Gestion des boucles

L'analyse réalisée par EVA doit être la plus précise possible pour garantir l'absence d'erreurs lors de l'exécution du code analysé. Une difficulté consiste à analyser les boucles (boucles for et boucles while) présentes dans le code à analyser. En effet, l'analyse par défaut réalisée par EVA produit souvent des résultats approximatifs car sans annotation spécifique, EVA ne peut pas déterminer le nombre d'itérations des boucles. Il est par conséquent nécessaire d'améliorer la précision de l'analyse des boucles :

- en indiquant à EVA le nombre maximal d'itérations dans une boucle. Cela permet à EVA d'analyser chaque itération indépendamment les unes des autres, mais a pour inconvénient d'augmenter le temps de calcul lorsque le nombre d'itération est important. Il est possible de traiter chaque boucle en les annotant séparément (`//@ loop unroll 10;` par exemple) ou en utilisant une option d'EVA qui fixe un nombre maximal d'itération pour l'ensemble du code à analyser (`-eva-auto-loop-unroll 10` par exemple) ;

5. `assert P` est interprété par Frama-C comme : vérifie que la propriété `P` est vraie au moment où l'assertion est utilisée

- en indiquant à EVA le nombre maximal d'états du code qu'il est possible de conserver en mémoire avant de devoir fusionner certains états. Par exemple, si le code présente une boucle de 10 itérations, comprenant une condition if / else, alors il faut indiquer à EVA de prendre en compte 20 états différents (2 états possibles pour chaque itération de la boucle). Ce traitement peut se faire de manière générique, en utilisant l'option `-eva-slevel n` (n représentant le nombre maximal d'états différents), ou par fonction `-eva-slevel-function f :n`, afin d'analyser plus finement le code tout en conservant un temps de calcul correct. L'utilisation de `slevel` permet également de remplacer l'utilisation de "unroll" ;
- en annotant chaque boucle à l'aide d'invariant (structure ACSL), ce qui permet d'indiquer à EVA une propriété valide avant l'entrée dans la boucle, pendant chaque itération et à la sortie de la boucle. L'utilisation d'invariant permet limiter l'approximation réalisée par EVA lors de l'analyse de la boucle. Un exemple d'annotation est le suivant : `//@ loop invariant 0 ≤ i ≤ 10 ;`

Le traitement des boucles par EVA dans le cadre de ce stage est détaillé dans le chapitre 4.2.2.

2.3.3 Utilisation de l'interface graphique

A l'aide de l'interface graphique, il est possible de visualiser les valeurs possibles de chacune des variables du code source analysé à l'aide du greffon EVA, comme illustré dans la figure suivante :

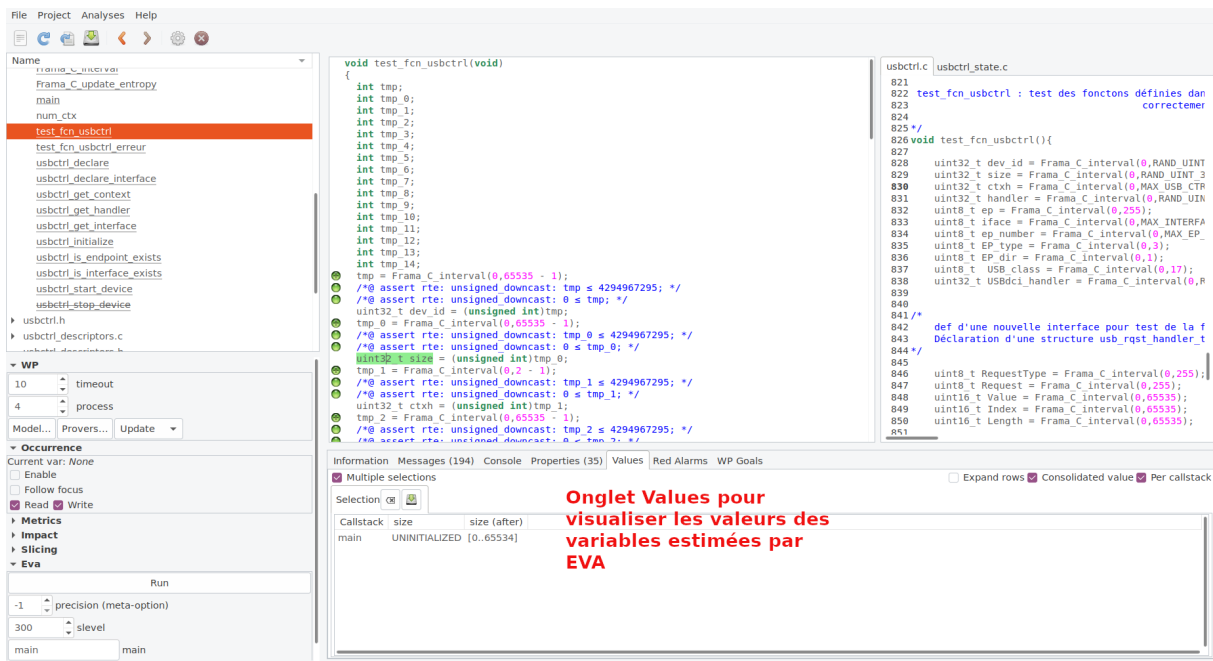


FIGURE 2.4 – Utilisation d'EVA dans l'interface graphique

Dans cet exemple, la variable `size` peut prendre l'ensemble des valeurs possibles en 0 et 65534. Cette propriété de l'interface graphique est très utile lors de la navigation dans le code source pour visualiser l'évolution des différentes variables, notamment à des fins de débogage.

2.4 WP

2.4.1 Présentation et objectifs

Le greffon WP (Weakest Precondition) est utilisé afin d'obtenir la preuve mathématique des propriétés du code analysé :

- des propriétés simples, comme l'absence de RTE ;
- des propriétés plus complexes, comme des propriétés de sécurité (ici mettre exemple) ou des propriétés fonctionnelles (ici mettre exemple)

WP est modulaire, c'est à dire qu'il est utilisé fonction par fonction dans Frama-C (à la différence d'EVA, qui nécessite un ou plusieurs points d'entrée dans le code pour le parcourir). WP utilise les spécifications du code à analyser afin de générer des conditions de vérification, qui sont elles mêmes vérifiées à l'aide de proveurs automatiques externes à l'outil Frama-C (Cf. 2.4.5). Si l'ensemble des conditions de vérification est prouvé, cela signifie que le programme respecte sa spécification. Il est important de noter l'importance de spécifier correctement le code : si les spécifications sont fausses, incomplètes ou trop générales, alors le respect de ces spécifications ne permet pas de garantir les propriétés fonctionnelles du code.

2.4.2 Logique de Hoare et plus faible précondition

Avant de détailler le fonctionnement de WP, il est nécessaire d'expliquer ce qu'est la logique de Hoare. Dans la logique de Hoare, un programme est considéré comme un transformateur d'états (un état étant représenté par les valeurs de l'ensemble des variables du programme). L'exécution d'un programme, ou d'une fonction du programme, a pour effet de transformer un état initial en un état final. La spécification d'un programme s'effectuera en formulant des propriétés sur ces deux états. En résumé, la logique de Hoare permet de décrire l'évolution d'un programme.

La logique de Hoare est utilisée afin de fournir un outil de vérification formelle des programmes sans les exécuter (analyse statique) et de décrire rigoureusement la sémantique des langages de programmation. La logique de Hoare permet d'introduire une notion importante, le triplet de Hoare, ainsi que deux définitions associées. Un triplet de Hoare est composé du programme P , de la pré-condition Pre (aussi nommée prédicat) et de la post-condition $Post$. Pre et $Post$ sont des formules logiques représentant des propriétés sur les variables du programme.

Le triplet de Hoare $\{Pre\} P \{Post\}$ a la signification suivante : Si Pre est satisfait et P termine alors $Post$ est satisfait après exécution de P . On parle dans ce cas de correction partielle.

Le triplet de Hoare $[Pre] P [Post]$ a la signification suivante : Si Pre est satisfait alors P termine et $Post$ est satisfait après exécution. On parle de correction totale.

L'utilisation de la logique de Hoare peut s'illustrer à travers l'exemple des contrats d'assurance :

- Les préconditions sont les primes payées par l'assuré
- Le programme P correspond à l'événement pour lequel le contrat d'assurance a été souscrit
- Les post-conditions sont les garanties apportées par l'assureur

Le greffon WP utilise la méthode de "plus faible précondition" pour garantir les propriétés fonctionnelles d'un programme. D'un point de vue mathématiques, WP permettant d'obtenir une correction totale du code et en utilisant le triplet de Hoare suivant : $[Pre] P [Post]$, cela signifie qu'à partir d'une annotation du code à analyser, par exemple une assertion $Post$ après un état P , la plus faible précondition Pre est la propriété Pre la plus simple qui doit être valide avant P de telle manière que $Post$ est vérifié après P . Cette procédure est définie par induction⁶ sur le système de preuve. Il est important de noter

6. est-ce que j'explique en une phrase le principe d'un raisonnement par induction (raisonnement par récurrence ?)

que WP cherche à fournir une correction totale du code analysé. Ce point sera notamment détaillé dans le chapitre 2.4.4 relatif au traitement des boucles avec WP.

2.4.3 Contrats de fonction

Les contrats de fonction sont le coeur de la vérification formelle réalisée à l'aide du greffon WP de Frama-C. Les contrats de fonction, définis en langage ACSL, permettent de définir les conditions d'exécution d'une fonction. Ces contrats comportent deux parties :

- Une ou plusieurs préconditions : les conditions supposées vraies en entrée de la fonction (conditions sur les variables, sur l'état de la mémoire (validité des pointeurs par exemple)). La preuve que celles-ci sont effectivement validées n'interviendra qu'aux points où la fonction est appelée ;
- Une ou plusieurs post-conditions : les conditions qui doivent être validées en sortie de la fonction (conditions sur la valeur de retour, sur l'état de la mémoire)

Les préconditions de fonctions sont introduites par la clause `requires` et les postconditions par la clause `ensures`. Un exemple de contrat de fonction est le suivant :

```
#include <limits.h>

/*@
    requires INT_MIN < val;
    ensures (val >= 0 ==> \result == val) &&
           (val < 0 ==> \result == -val);
*/
int abs(int val){
    if(val < 0) return -val;
    return val;
}
```

Dans cet exemple, la fonction spécifiée est une fonction qui retourne la valeur absolue d'un entier. La précondition porte sur la valeur minimale de l'entier qui doit être supérieure à `INT_MIN` (valeur définie dans le fichier `limits.h`) afin de ne pas avoir de débordement d'entier. La postcondition est simplement que le résultat soit égal à la valeur de l'entier s'il est positif ou nul, ou égal à son opposé s'il est strictement négatif.⁷

Les contrats de fonction permettent également de spécifier les effets de bords autorisés, c'est à dire de spécifier les variables et structures non locales qui sont modifiées par la fonction. Cette spécification se fait à l'aide de la clause `assigns` (qui est également une clause de postcondition). La clause `assigns` est très importante car, par défaut, WP considère qu'une fonction a le droit de modifier n'importe quel élément en mémoire. Il est donc nécessaire de déterminer les éléments qu'une fonction peut modifier afin d'avoir un contrat de fonction représentant de manière la plus précise possible son comportement. De la même manière, lorsque la fonction n'a aucun effet de bord, alors la clause `assigns \nothing` indique que la fonction ne modifie aucun élément en dehors des variables locales.

L'exemple suivant illustre un contrat de fonction avec une clause `assigns` :

```
/*@
    requires \valid(a) && \valid(b);
    assigns  *a, *b ;
    ensures  \old(*a) < \old(*b) ==> *a == \old(*b) && *b == \old(*a) ;
    ensures  \old(*a) >= \old(*b) ==> *a == \old(*a) && *b == \old(*b) ;
*/
void max_ptr(int* a, int* b){
    if(*a < *b){
        int tmp = *b ;
        *b = *a ;
    }
}
```

7. `\result` est une construction ACSL utilisée dans les spécifications afin de désigner le résultat d'une fonction

```

    *a = tmp ;
  }
}

```

Pour cette fonction, qui échange 2 valeurs pointées quand la valeur pointée par `a` est inférieure à celle pointée par `b`, le contrat de fonction spécifie que :

- les pointeurs `a` et `b` sont supposés valides en entrée de fonction ;
- si $*a < *b$, les valeurs pointées sont interverties⁸ ;
- si $*a \geq *b$, les pointeurs ne sont pas modifiés ;
- seuls les pointeurs `a` et `b` sont modifiés par la fonction (assigns `*a`, `*b`).

Enfin, les contrats de fonction permettent également de spécifier le comportement d’une fonction plus finement, notamment lorsque l’état de sortie d’une fonction dépend fortement des conditions d’entrée dans la fonction. Un cas typique est un test sur un pointeur : si le pointeur est `NULL` ou pas, le comportement de la fonction sera totalement différent. Ce cas peut être spécifié dans les contrats de fonction à l’aide du mot clé `behavior`, dont la structure est la suivante :

```

/*@
  behavior cpt1:
    assumes Pre_1;
    assigns ... ;
    ensures Post_1;

  behavior cpt2:
    assumes Pre_2;
    assigns ... ;
    ensures Post_2;

  ...

*/

```

Les comportements servent à spécifier les différents cas pour les postconditions. Chaque comportement a un nom. Pour un comportement donné, les différentes hypothèses à propos de l’entrée de la fonction sont introduites à l’aide du mot clé `assumes`. Les postconditions sont enfin introduites à l’aide du mot clé `ensures`. Il est également possible de vérifier le fait que les comportements sont disjoints (pour garantir le déterminisme) avec `disjoint behaviors` et complets (pour garantir que toutes les entrées possibles sont couvertes) avec `complete behaviors`. La vérification que les comportements sont complets et disjoints est en réalité très importante à vérifier lors de la spécification des fonctions lors du vérification de leurs comportements.

Le contrat de fonction de la valeur absolue devient ainsi, en utilisant les comportements et en ajoutant une clause `assigns` :

```

#include <limits.h>

/*@
  requires INT_MIN < val;

  behavior positif:
    assumes val >= 0 ;
    assigns \nothing ;
    ensures \result == val ;

  behavior negatif:

```

8. la structure `\old` dans les spécifications ACSL permet de faire référence à la valeur initiale d’un pointeur avant qu’il ne soit modifié.

```
        assumes val < 0 ;
        assigns \nothing ;
        ensures \result == -val ;

    complete behaviors;
    disjoint behaviors;

*/

int abs(int val){
    if(val < 0) return -val;
    return val;
}
```

2.4.4 Les boucles

La spécification des fonctions à l'aide des contrats de fonction est certes essentielle lors de l'utilisation de WP, mais elle ne suffit pas lorsque les fonctions contiennent des boucles. En effet, Pour prouver la correction totale d'un programme (telle que définie dans le chapitre 2.4.2, il faut d'une part prouver sa correction partielle, et d'autre part garantir la terminaison du code et donc s'assurer que les boucles s'arrêtent nécessairement lorsque les pré-conditions sont vérifiées.

Pour ce faire, l'utilisation de WP requiert donc, pour chacune des boucles présentes dans les fonctions que l'on souhaite prouver, d'exprimer :

- une propriété vérifiée avant l'entrée dans la boucle, à chaque itération de la boucle et en sortie de boucle. Cette propriété s'appelle un invariant de boucle, et s'introduit en ACSL à l'aide de `loop invariant` ;
- de même que pour les contrats de fonctions, les effets de bord de la boucle, à l'aide de `loop assigns` (comme pour les contrats de fonction, il sera nécessaire d'indiquer l'ensemble des éléments extérieurs à la boucle mais modifiés à l'intérieur de celle-ci) ;
- une expression fonction des variables du programme, positive lorsqu'on entre dans la boucle et qui décroît strictement à chaque itération de la boucle. Cette expression, appelée *variant* et introduite à l'aide d'un `loop variant`, constituera la condition d'arrêt de la boucle.

L'exemple suivant illustre la spécification d'une boucle pour l'utilisation de WP :

```
int main(){
    int i = 0;
    int h = 42;

    /*@
        loop invariant 0 <= i <= 30;
        loop assigns i;
        loop variant 30 - i;
    */
    while(i < 30){
        ++i;
    }
}
```

Dans cet exemple, la propriété qui doit être vérifiée avant, pendant et après la boucle est $0 \leq i \leq 30$. La variable `i` étant modifiée à chaque itération, `loop assigns i` est nécessaire pour prouver le comportement de la boucle. Enfin, la condition d'arrêt de la boucle est la valeur `loop variant 30 - i` positive à l'entrée de la boucle et qui décroît strictement à chaque itération de la boucle.

Nous disposons maintenant avec les contrats de fonction et la spécification des boucles de l'ensemble des éléments permettant d'analyser un code avec WP afin de vérifier ses propriétés fonctionnelles. Toutefois,

le calcul de la plus faible précondition avec WP n'est qu'une étape préliminaire à cette vérification, comme nous allons le voir dans le chapitre suivant.

2.4.5 Utilisation de prouveurs externes avec WP

WP calcule la plus faible précondition à partir des spécifications définies dans un contrat de fonction. Une fois ce calcul réalisé, des obligations de preuves sont générées par WP.

Les obligations de preuves générées par WP contiennent :

- les suppositions données dans les spécifications et celles déduites par WP dans son calcul de plus faible précondition ;
- les propriétés qui sont à vérifier.

Les obligations de preuve sont transformés en une formule logique puis transmises à des prouveurs automatiques externes l'outil Frama-C pour les résoudre. Toutefois, avant d'envoyer cette formule aux prouveurs, WP utilise le module Qed (interne au greffon WP) afin de simplifier les obligations de preuve lorsque c'est possible. Dans certains cas, ces simplifications sont suffisantes pour rendre triviales les obligations de preuve, qui sont donc directement satisfaites et non transmises aux prouveurs.

Il existe de nombreux prouveurs automatiques. Dans le cadre de ce stage les prouveurs suivants ont été utilisés :

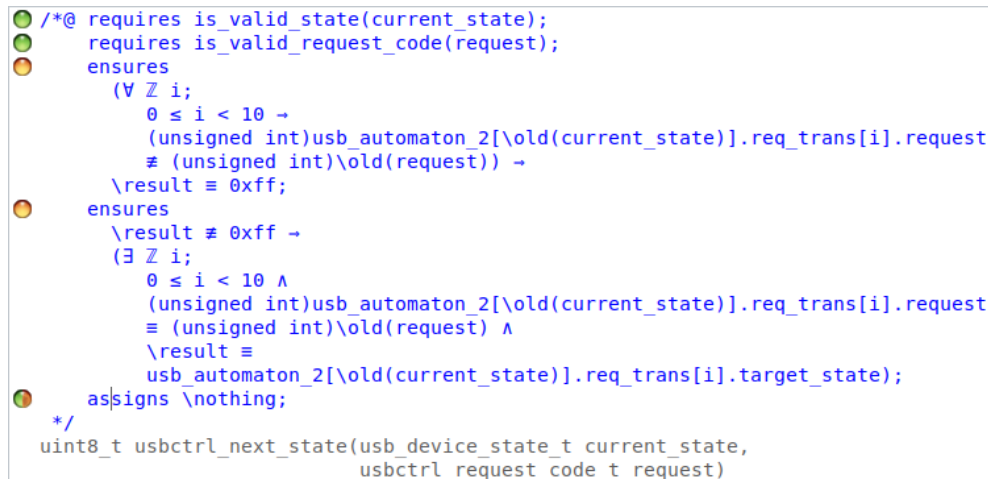
- Alt-Ergo (version 2.30), développé à l'INRIA et maintenu actuellement par OcamlPro ;
- CVC4 (Cooperating Validity Checker), développé par l'université de Stanford ;
- Z3, développé par Microsoft Research et le MIT.

Ces différents prouveurs sont utilisés à travers la plateforme why-3, externe elle aussi à Frama-C et développé par l'INRIA, l'université de Paris-Saclay et le CNRS.

2.4.6 Utilisation de l'interface graphique

De la même manière qu'avec EVA, il est possible d'utiliser WP en lignes de commande ou à travers l'interface graphique.

Comme précisé dans le chapitre 2.1.2, l'interface graphique permet d'abord de visualiser le résultat des preuves avec WP :



```

1  /*@ requires is_valid_state(current_state);
2  requires is_valid_request_code(request);
3  ensures
4      (∀ Z i;
5       0 ≤ i < 10 →
6         (unsigned int)usb_automaton_2[\old(current_state)].req_trans[i].request
7         ≠ (unsigned int)\old(request)) →
8       \result ≡ 0xff;
9  ensures
10     \result ≠ 0xff →
11     (∃ Z i;
12      0 ≤ i < 10 ∧
13      (unsigned int)usb_automaton_2[\old(current_state)].req_trans[i].request
14      ≡ (unsigned int)\old(request) ∧
15      \result ≡
16      usb_automaton_2[\old(current_state)].req_trans[i].target_state);
17  assigns \nothing;
18  */
19  uint8_t usbctrl_next_state(usb_device_state_t current_state,
20                           usbctrl_request_code_t request)

```

FIGURE 2.5 – Visualisation du résultat de WP avec l'interface graphique

Dans cet exemple, toutes les preuves n'ont pas été satisfaites (bullets orange et vert/orange). L'utilisation de l'interface graphique permet par ailleurs de relancer l'analyse de WP et des prouveurs automatiques sur certains contrats de fonction ou partie de contrats de fonction (vérification des effets de bord (clause assigns) ou des postconditions (clause ensures) par exemple. Les options utilisées en ligne de commande pour lancer la première preuve sont conservées, cela permet donc de travailler sur la preuve d'une fonction sans devoir relancer l'analyse de l'ensemble du code source. Il est toutefois possible de modifier quelques options de WP dans l'interface graphique (les prouveurs automatiques utilisés, le temps de calcul maximal pour chaque preuve (timeout), le modèle mémoire⁹), comme le montre la figure ci-dessous :

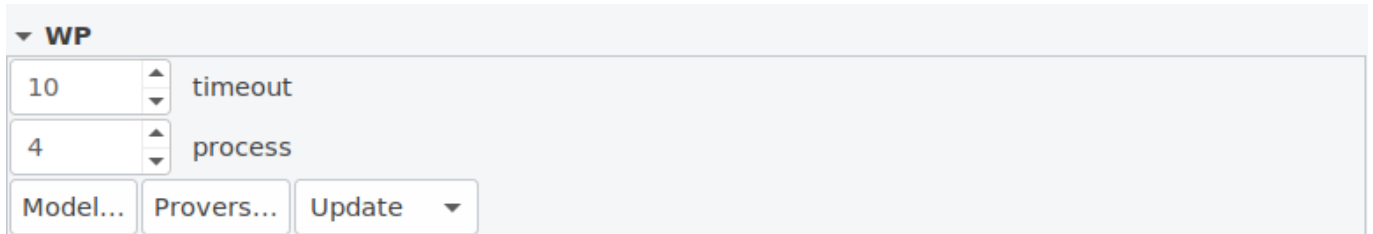


FIGURE 2.6 – Options de WP avec l'interface graphique

9. le modèle mémoire est expliqué plus en détail dans le chapitre 4.2.3 relatif aux options de WP dans le cadre de ce stage

Enfin, l'interface graphique rend également possible une analyse fine du résultat des preuves, avec l'onglet "WP Goals" comme montré dans la figure ci-dessous :

The screenshot displays a code editor with C code and a table of proof results. The code includes assertions and state transitions related to USB device states. The table below summarizes the proof results for various goals.

Module	Goal	Model	Qed	Script	Alt-Ergo 2.3.0	CVC4 1.6	CVC4 1.6 (cou)
is_std_requests_allowed	Complete behaviors 'false', 'true'	Typed (Ref)	—	—	●	●	●
is_std_requests_allowed	Disjoint behaviors 'false', 'true'	Typed (Ref)	—	—	●	●	●
is_std_requests_allowed	Assertion	Typed (Ref)	—	—	▼	▼	▼
is_std_requests_allowed	Assertion	Typed (Ref)	—	—	●	●	●
is_std_requests_allowed	Assigns nothing (exit)	Typed (Ref)	●	▶			
is_std_requests_allowed	Assigns nothing (exit)	Typed (Ref)	●	▶			
is_std_requests_allowed	Assigns nothing (return)	Typed (Ref)	●	▶			
is_std_requests_allowed	Assigns nothing (return)	Typed (Ref)	●	▶			

FIGURE 2.7 – Visualisation des résultats des preuves avec l'interface graphique

Cet onglet permet de visualiser l'ensemble des résultats des preuves pour une fonction ou une assertion. Les preuves satisfaites sont identifiées par une bulle verte positionné soit au niveau du module Qed soit pour différents prouveurs automatiques quand ceux-ci ont été utilisés. Quand une preuve n'est pas satisfaite (dans l'exemple de la figure, une assertion n'a pas été prouvée), les prouveurs n'ayant pas pu satisfaire la preuve sont identifiés. Il est possible d'examiner en détail la preuve ayant échoué¹⁰ :

Information Messages (119) Console Properties (85) Values Red Alarms WP Goals

Global All Results Provers... Clear

Raw Obligation Decimal Real Non Proved Property Tactics

No Script

Goal Assertion:
 Let $x = \text{Mint_0}[\text{shiftfield_F206_usbctrl_context_state}(\text{ctx_0})]$.
 Assume {
 Type: is_uint32(tmp_0) /\ is_uint32(tmp_0_0) /\ is_uint8(x).
 (* Heap *)
 Have: (region(ctx_0.base) <= 0) /\ linked(Malloc_0).
 (* Pre-condition *)
 Have: valid_rd(Malloc_0, ctx_0, 779).
 (* Call 'usbctrl_get_state' *)
 Have: (((null = ctx_0) -> (tmp_0 = 9))) /\
 (((null != ctx_0) -> (P_is_valid_state(x) -> (x = tmp_0)))).
 If tmp_0 = 6
 Else {
 Have: (ta_tmp_0=false).
 (* Call 'usbctrl_get_state' *)
 Have: (((null = ctx_0) -> (tmp_0_0 = 9))) /\
 (((null != ctx_0) -> (P_is_valid_state(x) -> (x = tmp_0_0)))).
 If tmp_0_0 = 7
 Else {
 Have: (ta_tmp_1=false).
 (* Call 'usbctrl_get_state' *)
 Have: (null != ctx_0) /\ ((P_is_valid_state(x) -> (x = 8))).
 }
 }
 }
 Prove: $(x = 6) \vee (x = 7) \vee (x = 8)$.

Prover Z3 4.8.4 (noBV): Unknown (Qed:32ms) (cached).
 Prover Z3 4.8.4 (counterexamples): Unknown (Qed:32ms) (cached).
 Prover Z3 4.8.4: Unknown (Qed:32ms) (cached).
 Prover CVC4 1.6 (counterexamples): Timeout (Qed:32ms) (10s) (cached).
 Prover CVC4 1.6: Timeout (Qed:32ms) (30s) (cached).
 Prover Alt-Ergo 2.3.0: Timeout (Qed:32ms) (30s) (cached).

FIGURE 2.8 – Exemple d'une preuve non satisfaite

10. toutefois, l'analyse des preuves est relativement complexe et difficile à exploiter

2.5 Stratégie d'utilisation de Frama-C

Frama-C est un outil modulaire, présentant de nombreuses fonctionnalités de par ses nombreux greffons et les options associées à ceux-ci. Plusieurs stratégies d'analyse sont possibles (en lignes de commande, avec l'interface graphique, en utilisant en parallèle les différents greffons ou au contraire en les utilisant successivement...).

Dans le cadre de ce stage, la stratégie d'analyse mise en place est la suivante :

- dans un premier temps, Frama-C est lancé sans greffon (avec simplement le noyau) en ligne de commande, afin de parser le code source à analyser. Cela permet de vérifier que Frama-C est capable de parcourir l'ensemble du code source à analyser y compris les fichiers inclus dans le code lors du preprocessing ;
- dans un deuxième temps, le greffon EVA est utilisé, d'abord avec les options par défaut puis en raffinant au fur et à mesure de façon à gagner en précision. L'objectif est d'avoir un taux de couverture maximal avec EVA de façon à détecter le plus de RTE possibles. Les RTE détectées sont corrigées avant de passer à l'étape suivante ;
- dans un dernier temps, à la suite de l'analyse réalisé avec EVA, le greffon WP est utilisé afin de vérifier les spécifications fonctionnelles du code à analyser, en s'aidant du résultat de l'analyse effectuée avec EVA. En effet, WP ne génère pas d'obligation de preuve pour vérifier l'absence de RTE. WP suppose au contraire que le code ne comprend pas de RTE. L'utilisation d'EVA avant WP permet de pallier à ce problème et d'être sûr que l'analyse de WP porte sur un code ne comprenant pas de RTE¹¹ et les assertions validés par EVA (assertions automatiques ou assertions ajoutées manuellement) peuvent être utilisées par WP et par les prouveurs automatiques afin de satisfaire les obligations de preuve de WP.

La figure suivante résume la stratégie d'utilisation de Frama-C mise en œuvre dans ce stage :

11. pour rappel, l'analyse d'EVA se base sur une sur-approximation du code ce qui garantit l'absence de faux négatifs dans le code à analyser - analyse correcte d'EVA

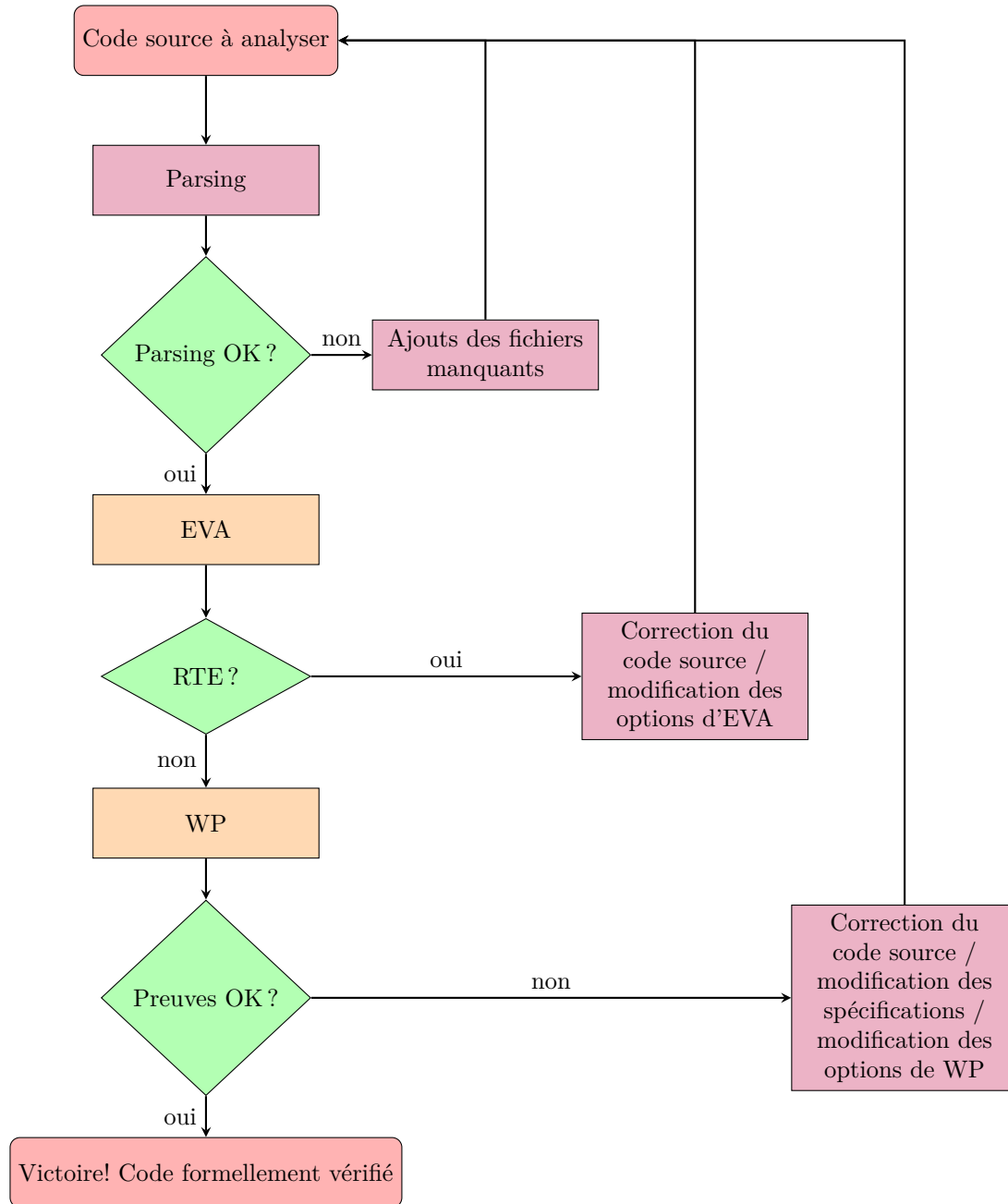


FIGURE 2.9 – Stratégie d'utilisation de Frama-C

Chapitre 3: Projet analysé

3.1 Présentation de WooKey

L'ANSSI développe depuis 2014 un projet de clé usb chiffrante sécurisée, le projet WooKey. Ce projet, open-source et open-hardware, s'inscrit dans le cadre d'une volonté croissante de la part de la communauté académique de sécuriser le protocole USB, notamment depuis la découverte et la publication de l'attaque BadUSB¹

L'objectif de WooKey est de fournir les fonctionnalités suivantes :

- La protection des données des utilisateurs. L'ensemble des données sont chiffrées, ce qui permet d'assurer leur protection en confidentialité²
- L'utilisateur doit être présent lorsque les données sont déchiffrées (authentification forte)
- Le logiciel du périphérique doit pouvoir être mis à jour de manière robuste, les fichiers de mis à jour doivent en particulier être authentifiés et leur intégrité vérifiée avant toute mise à jour
- Les attaques logicielles doivent être prises en compte, afin de garantir qu'un attaquant ne puisse pas utiliser la surface d'attaque du logiciel (l'USB par exemple) pour obtenir un accès privilégié à la plateforme ainsi qu'un accès aux clés cryptographiques utilisées pour le chiffrement des données de l'utilisateur

Concrètement, la plateforme WooKey consiste en un micro-contrôleur 32 bits, de type STM32, se comportant comme un périphérique de stockage de masse. Les principales caractéristiques de sécurité de WooKey sont les suivantes :

- Mise en oeuvre du concept de défense en profondeur,
- Logiciel de mise à jour sécurisé
- Double facteur d'authentification, utilisant une carte à puce et les principes de cryptographie les plus récents
- Une architecture logicielle modulaire, avec le cloisonnement des différents modules, présentée dans la figure suivante :
 - Les modules TOKEN et PIN pour l'authentification double facteur,
 - Le module CRYPTO pour le chiffrement des données transférées entre le périphérique et la machine hôte,
 - Le module SDIO (Secure Digital Input/Output), pour le stockage des données chiffrées sur une carte SDIO,
 - Le module USB, dans lequel le protocole USB 2.0 est mis en oeuvre. C'est dans ce module que se trouve la bibliothèque analysée dans le cadre de ce stage, décrite dans le paragraphe suivant.

1. L'exploitation d'une telle attaque rend possible, par exemple, la programmation de frappes sur un clavier afin d'injecter une charge dans une machine cible à partir d'un périphérique USB différent d'un clavier

2. la protection des données en intégrité n'est pas encore garantie (à confirmer)

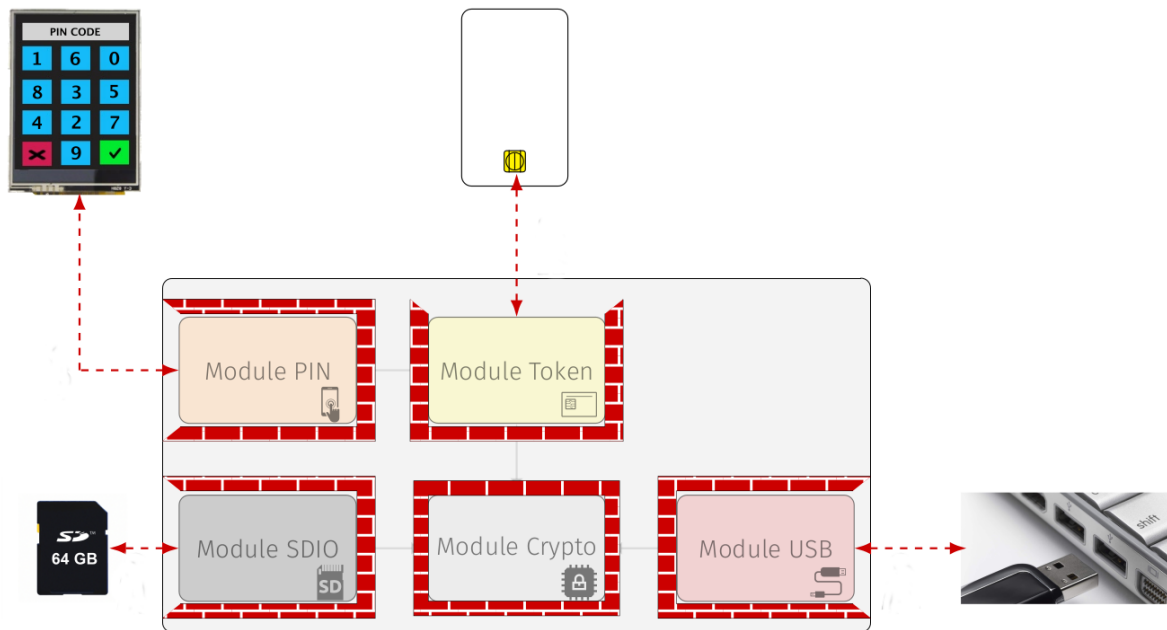


FIGURE 3.1 – Architecture logicielle de WooKey

3.2 Présentation de la bibliothèque USBctrl

Avant de détailler le fonctionnement de la bibliothèque USBctrl analysée dans le cadre de ce stage, il est nécessaire de présenter le protocole USB et ses spécifications.

3.2.1 Généralités sur le protocole USB et spécifications de l'USB 2.0

L'USB (Universal Serial Bus) est, comme son nom l'indique, un protocole de communication série entre entités. Plusieurs versions sont actuellement disponibles, de l'USB 1.0, la plus ancienne version, jusqu'à la version USB 4.0, la plus récente). WooKey peut actuellement fonctionner avec des versions USB inférieures ou égales à 2. Une des différences principales entre les différentes versions est la vitesse de transfert des données entre un périphérique USB et une machine hôte (entre 1,5 Mbits/s et 480 Mbits/s dans le cadre de WooKey).

Du point de vue utilisateur, le bus USB se présente sous la forme d'une architecture étoilée et pyramidale, l'hôte se trouvant au centre du réseau, et les périphériques à l'extérieur. L'intérêt principal de ce bus est le fait qu'un grand nombre de périphériques (jusqu'à 126) peuvent être connectés simultanément au même hôte, et qu'à tout moment, il est possible de les débrancher et de les rebrancher sans redémarrer l'hôte. La figure suivante représente cet exemple d'architecture USB :

Les périphériques USB peuvent être regroupés en classes et sous-classes USB, telles que, par exemple :

- les claviers et les souris, faisant partie de la classe Human Interface Device (HID),
- Les imprimantes,
- Le stockage de masse (MSC),
- Les lecteurs de cartes à puce.

Le protocole USB est enfin caractérisé par une machine à états, décrite dans les spécifications de l'USB 2.0 en référence XX. Cette machine à états est présentée dans la figure suivante :

3.2.2 Rôle de la bibliothèque USBctrl dans le module USB de WooKey

Chaque périphérique USB 2.0 nécessite la mise en place d'une structure de contrôle afin de pouvoir initialiser la communication avec l'hôte, négocier les propriétés du canal de communication USB mis en

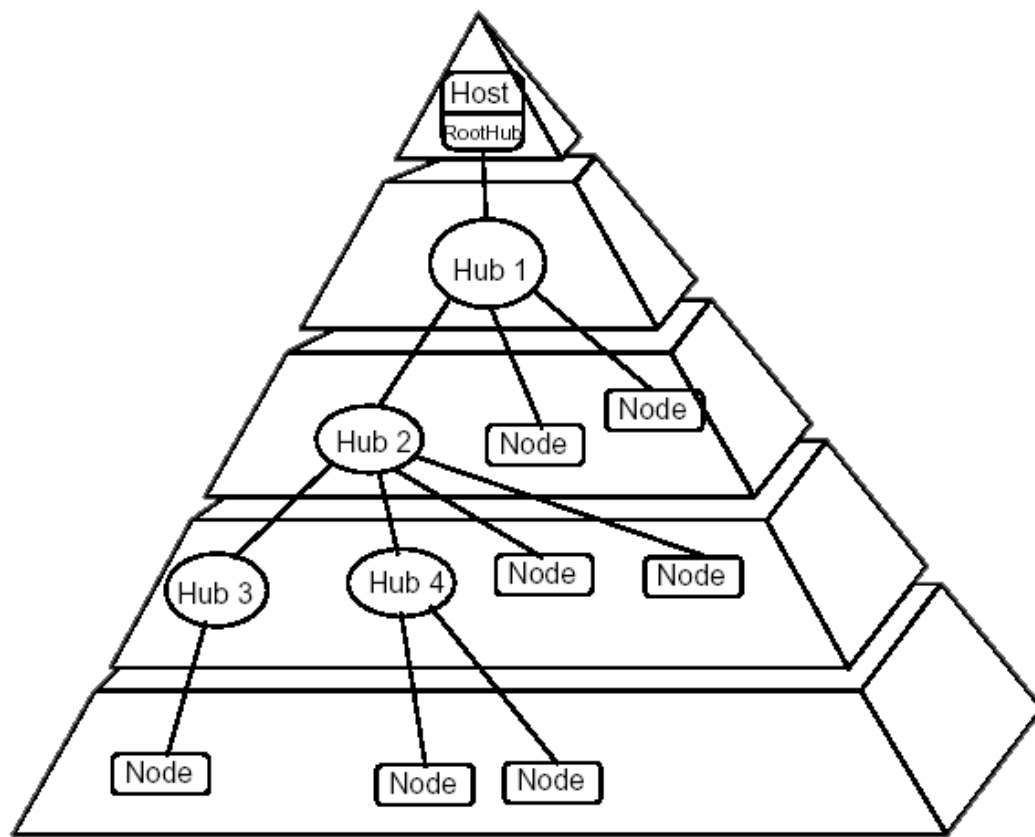


FIGURE 3.2 – Exemple d'architecture USB

place avec l'hôte et déclarer les capacités du périphérique (vitesse de transfert des données par exemple). C'est ce rôle que joue la bibliothèque USBctrl de WooKey. Elle permet par ailleurs de gérer la machine à états de l'USB 2.0, sans nécessiter d'actions complexes de la part des couches plus élevées du périphérique et, surtout, réalise l'abstraction du driver USB pour permettre une portabilité complète, quelque soit la classe USB ou le driver USB du périphérique.

Le contrôle du protocole USB réalisé par la bibliothèque USBctrl est réalisé en manipulant des structures caractéristiques du protocole USB :

- les points de terminaison (ou endpoint), qui sont des canaux de communication à travers lesquels les données sont transmises. Chaque point de terminaison a un type (comment sont formatées les données) et une direction (depuis ou vers le périphérique). Les points de terminaison sont à sens unique, les données pouvant soit être reçues, soit être transmises à travers eux. La bibliothèque USBctrl utilise le point de terminaison 0, à travers duquel sont transmises les données permettant la configuration du protocole USB. D'autres points de terminaison peuvent être utilisés par la suite, afin d'échanger des données entre le périphérique et l'hôte, mais ces points sont utilisés par les couches applicatives du module USB et non par la bibliothèque USBctrl.
- les interfaces qui :
 - définissent la classe et la sous-classe USB des couches applicatives en interface avec la bibliothèque USBctrl,
 - déterminent si la configuration USB peut gérer une ou plusieurs interfaces (par exemple, un périphérique USB avec 2 ports USB peut avoir deux classes USB différentes qui peuvent être gérées au sein d'une même configuration),
 - comprennent une liste de points de terminaison associés aux interfaces

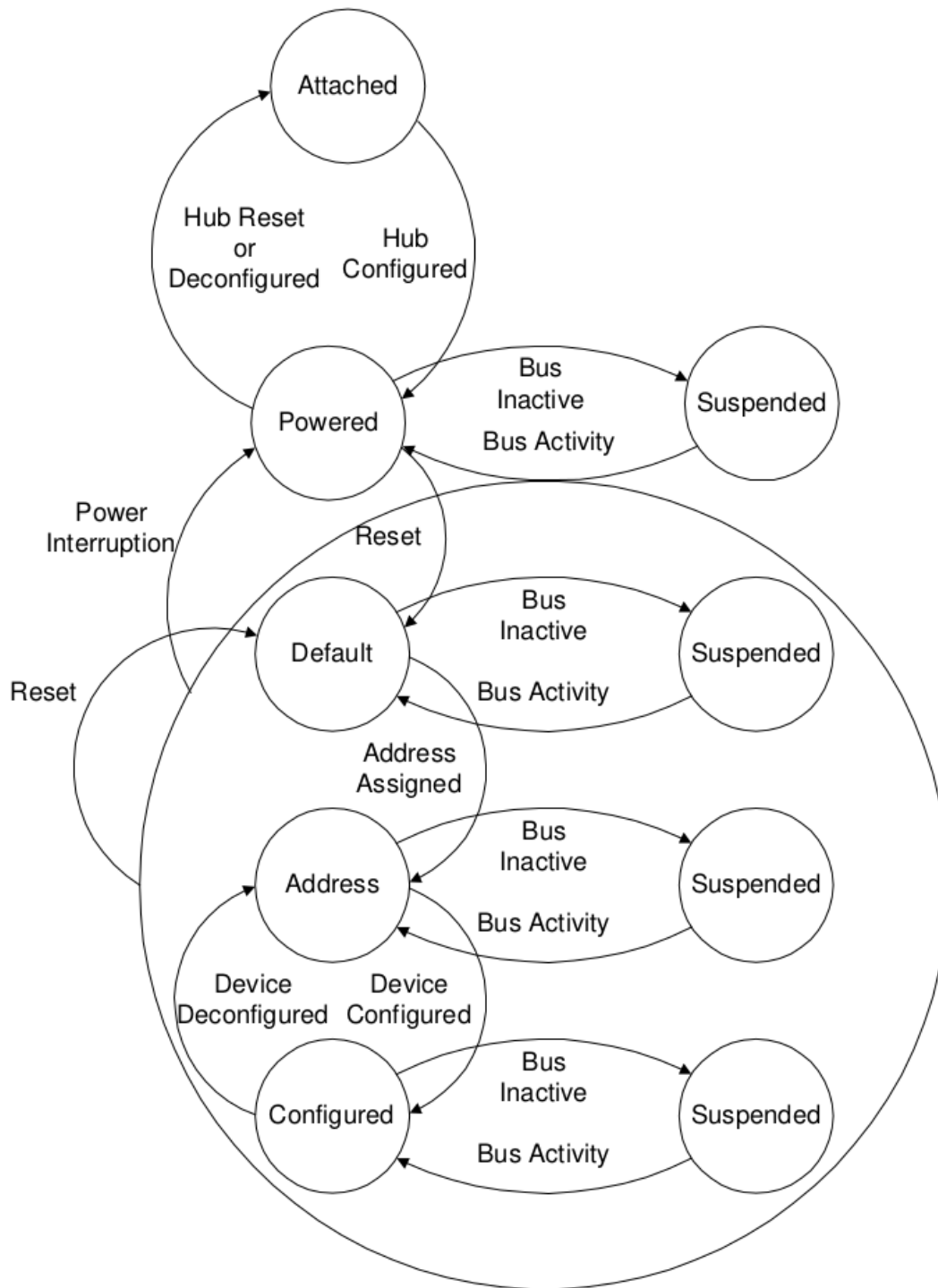


FIGURE 3.3 – Machine à états de l'USB 2.0

- le contexte USB, qui permet de gérer le point de terminaison de contrôle ainsi que les interfaces définies dans le périphérique. Ce contexte est caractérisé par un identifiant unique, par une adresse, par un état (au sens de la machine à état des spécifications de l'USB 2.0) et par le nombre des configurations USB du périphérique (un contexte peut avoir une ou plusieurs configurations différentes, et un périphérique USB peut avoir plusieurs contextes)

La figure suivante illustre le positionnement de la bibliothèque USBctrl dans la partie logicielle du module

USB de WooKey. Cette figure permet notamment de visualiser les interfaces entre la bibliothèque USBctrl et les couches applicatives et entre la bibliothèque USBctrl et le driver USB.

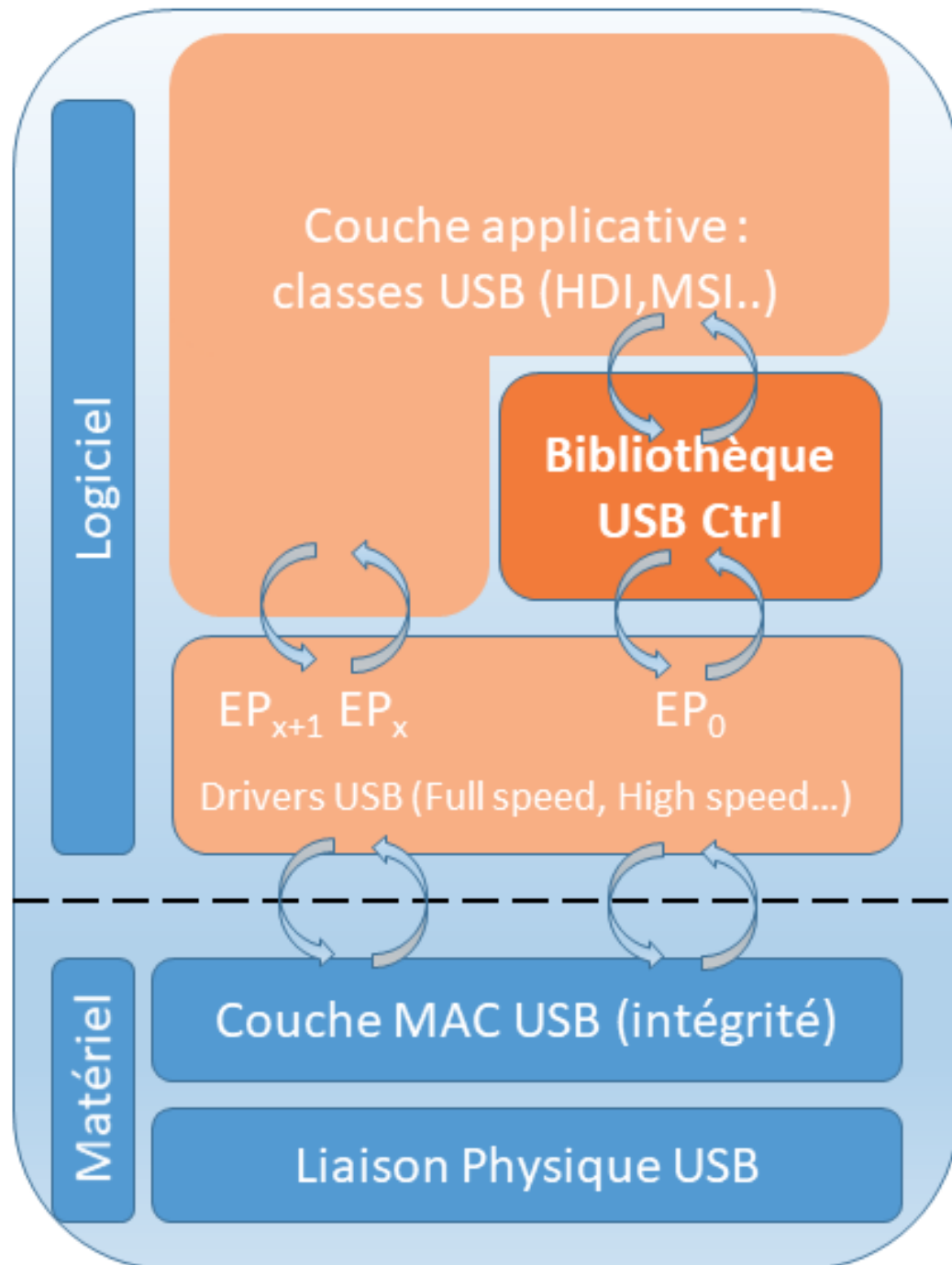


FIGURE 3.4 – Schéma logiciel de la bibliothèque USBctrl

3.3 Code source de la bibliothèque USBctrl

3.3.1 Description

La bibliothèque USBctrl se compose de 5 fichiers, pour environ 2000 lignes de code, dans lesquels sont implémentées une quarantaine de fonctions assurant les fonctionnalités de la bibliothèque présentées dans le chapitre précédent :

- le fichier `usbctrl.c`, comprenant l'API de la bibliothèque USBctrl : déclaration et initialisation du contexte USB, déclaration des interfaces, des points de terminaison et démarrage du périphérique USB,
- Le fichier `usbctrl_state.c`, comprenant les fonctions relatives à la gestion de la machine à états de l'USB 2.0,
- Le fichier `usbctrl_handlers.c`, comprenant les fonctions relatives à la gestion de certains événements tels que la réception d'un signal de reset en provenance de l'hôte, ou la réception ou l'émission de données à travers l'endpoint de contrôle (EP0),
- Le fichier `usbctrl_descriptors.c`, avec une seule fonction permettant de gérer la structure de données comprenant les caractéristiques des différentes configurations USB du périphérique,
- Le fichier `usbctrl_requests.c`, dont les fonctions permettent de répondre aux requêtes émises par l'hôte à destination du périphériques USB (par exemple, demande d'information sur la configuration du périphérique³)

Comme expliqué dans le paragraphe précédent, la bibliothèque USBctrl réalise l'abstraction des drivers USB à travers des alias dans les différents fichiers de la bibliothèque. Cela permet à la bibliothèque de pouvoir être utilisée avec n'importe quel driver USB et de garantir sa portabilité, tout en évitant d'avoir une fonction contrôle pour chaque driver (contrainte de quantité de mémoire disponible dans le périphérique). Néanmoins, il est apparu très tôt dans le cadre de ce stage que la vérification formelle de la bibliothèque USBctrl, pour être précise, nécessitait de vérifier formellement les fonctions du driver appelées par la bibliothèque. Par conséquent, une partie du driver USB high speed présent dans WooKey a été analysé⁴

Le driver high speed se compose de 4 fichiers, pour environ 2000 lignes de code. Ses différentes fonctions permettent notamment :

- De déclarer le contexte USB du driver USB et de le manipuler (initialisation, configuration...),
- De gérer le transfert de données entre le driver USB et l'hôte,
- De gérer les interruptions provenant de l'hôte,
- De lire ou d'écrire dans les registres mémoires du périphériques (adresses codées en dur, dépendantes du driver USB et du périphérique).

3.3.2 Rappel sur le langage C

La bibliothèque USBctrl analysée lors de ce stage est écrite en langage C. Ce langage de programmation est un langage de bas niveau, performant, qui permet aux programmeurs d'avoir un contrôle étendu sur leurs programmes, mais qui ne dispose pas des mécanismes de sécurité des langages de programmation plus récents comme Rust, en particulier pour la gestion de la mémoire. En effet, en C, la gestion de la mémoire est laissée à la charge du programmeur (allocation, désallocation, vérification de l'absence de buffer overflow...). Il est important de rappeler que la majeure partie des vulnérabilités logicielles découvertes ces dernières années sont dues à l'exploitation de buffer overflow.

De plus, le standard du langage C ne spécifie pas l'ensemble des comportements du code, et donc certains

3. les différentes requêtes auxquelles la bibliothèques doit pouvoir répondre sont définies dans les spécifications de l'USB 2.0

4. Deux drivers USB sont codés dans WooKey, le 2ème étant le driver full speed. Le driver high speed étant plus répandu, le choix s'est porté sur lui

restent indéfinis ou non spécifiés ce qui peut également mener à des RTE. A ce sujet, l'ANSSI a publié début 2020 un guide définissant un ensemble de règles, de recommandations et de bonnes pratiques dédiées aux développements sécurisés en langage C.

Chapitre 4: Utilisation de FramaC sur la bibliothèque USBctrl

4.1 Prise en main de Frama-C

4.1.1 Installation

L'installation de la plateforme Frama-C (version 20.0 Calcium) a été réalisée à l'aide du gestionnaire de paquet opam, qui permet d'avoir un environnement compatible avec l'utilisation de Frama-C. Pour compléter l'installation de Frama-C, des prouveurs externes ont également été installés, dans leur dernière version à l'exception d'Alt-ergo :

- Alt-ergo version 2.3.0¹,
- CVC4 version 1.7
- Z3 version 4.8.6

4.1.2 Tutorial

La première partie du stage a consisté à prendre en main Frama-C, les greffons EVA et WP ainsi qu'à apprendre les bases du langage ACSL. À ce titre, j'ai suivi plusieurs tutoriels :

- pour WP et ACSL, j'ai suivi un tutoriel en ligne (<https://zestedesavoir.com/tutoriels/885/introduction-a-la-preuve-de-programmes-c-avec-frama-c-et-son-greffon-wp>) détaillant l'utilisation de WP mais présentant également de nombreux exercices sur des contrats de fonction simples, réalisés en ACSL. De plus, le manuel en référence XX (ACSL by Example) présente de nombreux exemples de complexité croissante permettant de prendre en main ACSL et de monter en compétence.
- Pour EVA, j'ai suivi un tutoriel en ligne (<http://blog.frama-c.com/index.php?post/2017/03/07/A-simple-EVA-tutorial>) ainsi que le tutoriel présenté dans le manuel d'EVA (référence XX). Ces tutoriels expliquent la prise en main du greffon EVA sur des exemples simples, donnent quelques conseils génériques d'analyse et présentent notamment l'utilisation de l'interface graphique avec EVA,

En complément de ces différents tutoriels, chaque greffon de Frama-C dispose d'un manuel utilisateur (XX et XX) présentant la plupart des options pouvant être utilisées avec les greffons. Toutefois, les exemples donnés ne sont pas toujours bien illustrés, et toutes les options ne sont pas présentes. Il est ainsi nécessaire de se référer à l'aide en ligne de commande, installée en même temps que Frama-C.

4.2 Utilisation de Frama-C sur la bibliothèque USBctrl

La stratégie suivie pour l'analyse de la bibliothèque USBctrl avec Frama-C a été présentée dans le chapitre 2.5. Elle comporte 3 étapes, détaillées dans les paragraphes ci-dessous :

1. la dernière version est la 2.3.2, mais il est conseillé d'utiliser la version 2.3.0 pour des raisons de compatibilité avec why3, la plateforme exécutant les différents solveurs externes

- Parsing du code source de la bibliothèque USBctrl,
- Analyse avec EVA,
- Analyse avec WP.

4.2.1 Parsing du code source de la bibliothèque USBctrl de WooKey

La première étape de l'analyse de la bibliothèque USBctrl a consisté à vérifier que Frama-C pouvait parcourir l'ensemble des fichiers de la bibliothèque ainsi que ceux utilisés par la bibliothèque (fichiers du driver, fichiers de configuration...). Frama-C a été utilisée avec la ligne de commande suivante :

```
1 frama-c usbctrl.c usbctrl_descriptors.c usbctrl_handlers.c usbctrl_requests.c
2 usbctrl_state.c framac/include/driver_api/usbotghs_frama.c -c11
3 -machdep x86_32 -no-frama-c-stdlib -cpp-extra-args="-nostdinc -I framac/include "
```

Cette ligne de commandes est constituée :

- des fichiers qui seront analysés par Frama-C : les 5 fichiers de la bibliothèque USBctrl et 1 fichier d'un driver USB²
- d'options :
 - c11 car la bibliothèque USBctrl comporte du code utilisant le standard ANSI c11. Il est nécessaire de préciser cette option car, nativement, Frama-C utilise les standards c90 et c99. Sans cette option, Frama-C ne peut donc pas analyser du code C écrit dans le standard c11,
 - machdep x86_32 afin d'indiquer à Frama-C que la machine sur laquelle est censée s'exécuter le code est une machine 32 bits (l'architecture de WooKey est une architecture 32 bits),
 - no-frama-c-stdlib pour indiquer à Frama-C de ne pas utiliser la bibliothèque standard du C généralement installée sur toutes les distributions Linux. En effet, WooKey dispose de sa propre version de cette bibliothèque,
 - cpp-extra-args="-nostdinc -I framac/include", afin d'indiquer à Frama-C dans quel répertoire se trouvent les fichiers nécessaires pour compiler les fichiers analysés (bibliothèque C de WooKey, fichiers de configuration...).

4.2.2 Utilisation d'EVA

Une fois le parsing des fichiers réalisé avec Frama-C, la deuxième étape a consisté à utiliser EVA.

Stratégie d'analyse

L'analyse des fichiers avec EVA nécessite un point d'entrée dans le code source, à partir duquel EVA peut analyser les différentes fonctions du code source. Par défaut, ce point d'entrée est la fonction main du code analysé, mais ce n'est pas obligatoire et il est possible de définir n'importe quelle fonction comme point d'entrée en précisant cette fonction dans les options d'EVA. Une fois un point d'entrée dans le code défini, l'analyse avec EVA peut être effectuée. Deux critères sont importants pour estimer la pertinence de l'analyse réalisée :

- le taux de couverture du code analysé
- la précision de l'analyse réalisée

L'objectif est d'obtenir un taux de couverture du code maximal, avec la meilleure précision possible. De cette manière, il est possible de garantir à l'absence de RTE avec EVA, pour des expressions simples, car EVA réalise une analyse correcte³. La figure suivante représente l'objectif recherché de l'analyse réalisée avec EVA :

2. Le driver high speed codé dans WooKey se compose de 4 fichiers, pour des raisons pratiques j'ai choisi de concaténer ces différents fichiers dans un seul fichier, ce qui n'a aucune conséquence sur l'analyse réalisée avec Frama-C

3. La notion de correction est définie dans le chapitre 1. Cela signifie que l'analyse d'EVA ne comporte pas de faux négatifs, toutes les erreurs potentielles sont détectées

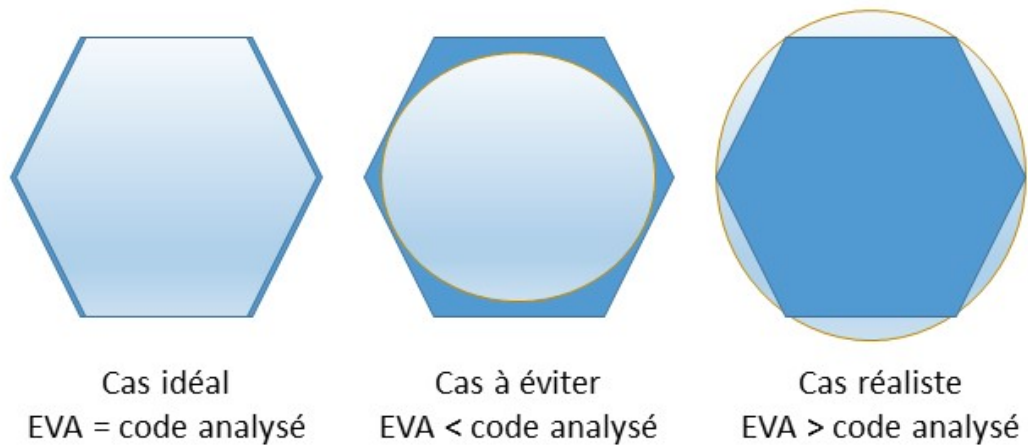


FIGURE 4.1 – Schéma d'analyse avec EVA

Le premier cas représente le cas idéal : l'interprétation abstraite d'EVA correspond exactement au code source analysé.

Dans le deuxième cas, le taux de couverture du code analysé n'est pas suffisant, et l'interprétation abstraite d'EVA ne permet pas de couvrir tout le code. Il existe donc des parties du code qui ne sont pas analysées par EVA, et il n'est donc pas possible de conclure à l'absence de RTE (même si EVA est correct).

Enfin, dans le dernier cas, qui correspond à l'objectif visé, le taux de couverture du code est suffisant et l'interprétation d'EVA réalise une sur-approximation du code analysé. Combiné au caractère correct d'EVA, il est possible dans ce cas de conclure à l'absence d'erreur dans le code analysé.

Le taux de couverture du code analysé est indiqué dans le résumé de l'analyse d'EVA, comme illustré dans la figure suivante :

```
[eva:summary] ===== ANALYSIS SUMMARY =====
-----
86 functions analyzed (out of 158): 54% coverage.
In these functions, 2020 statements reached (out of 2481): 81% coverage.
```

FIGURE 4.2 – Taux de couverture - EVA

Dans cet exemple, 54% de l'ensemble des fonctions du code source sont analysés : 86 fonctions sur 158, en incluant l'ensemble des fichiers inclus, qui ne sont pas forcément à analyser). Le taux de couverture des fonctions analysés est alors de 81% dans cet exemple. Le taux de couverture global du code doit donc s'évaluer à partir du nombre de fonctions analysées, et de leur couverture. Par exemple, si une seule fonction est analysée parmi une quarantaine de fonction, même si elle est analysée à 100%, le taux de couverture sera faible. De même, si 100% des fonctions sont analysés mais le taux de couverture de ces fonctions ne dépasse pas 10%, cela signifie qu'EVA n'analyse quasiment pas ces fonctions, même si EVA y rentre.

Par ailleurs, le taux de couverture du code peut être vérifié à l'aide de l'interface graphique de Frama-C, comme le montre la figure suivante :



FIGURE 4.3 – Visualisation des fichiers analysés avec EVA - Interface graphique

Sur cette figure, les fichiers et fonctions que Frama-C peut analyser sont indiqués sur la partie gauche. Les fonctions et fichiers non analysés par EVA sont barrés.

De plus, il est possible de visualiser pour une fonction donnée sa couverture dans l'onglet principal, comme illustré dans la figure suivante :

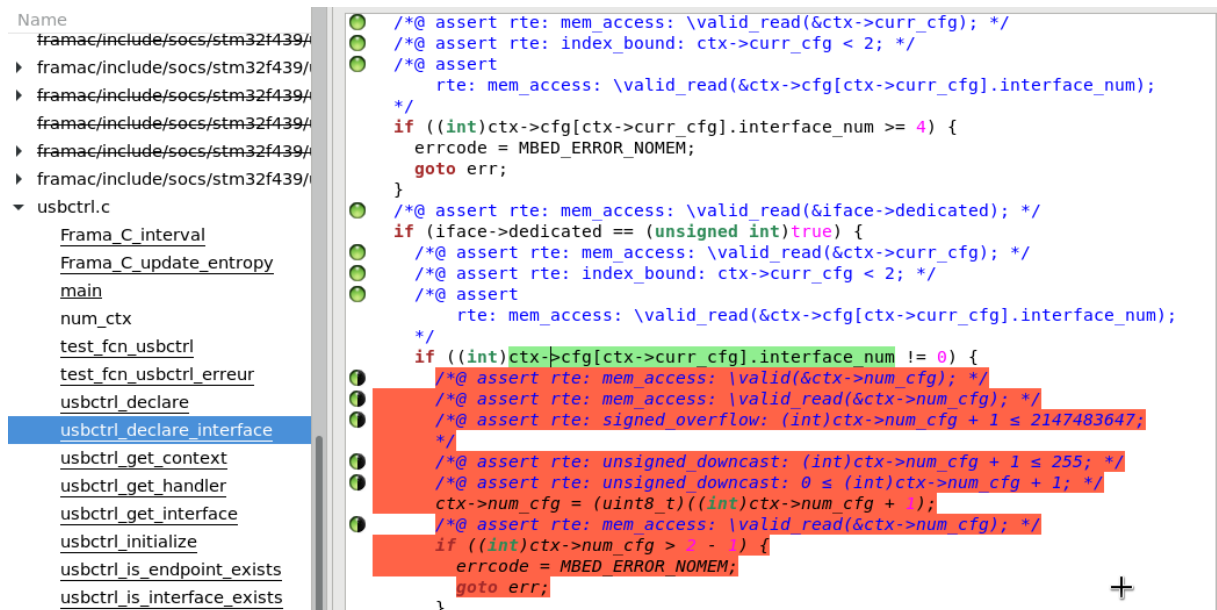


FIGURE 4.4 – Fonction analysée avec EVA - Interface graphique

Dans cet exemple, le contexte d'appel de la fonction analysée, `usbctrl_declare_interface`, ne permet pas d'entrer dans une structure conditionnelle. Pour augmenter le taux de couverture, il faut modifier le point d'entrée de l'analyse avec EVA en ajoutant les fonctions manquantes et en jouant sur les paramètres d'entrées afin par exemple de parcourir l'ensemble des embranchements possibles dans une fonction et en incluant les cas d'erreur le cas échéant.

Le deuxième paramètre important pour l'analyse avec EVA est la précision de l'analyse. En effet, si l'interprétation réalisée avec EVA n'est pas suffisamment précise, il ne sera pas possible d'obtenir l'ensemble des valeurs possibles pour toutes les variables du code analysé, et par conséquent il ne sera pas possible de conclure en l'absence d'erreur. En contrepartie, une plus grande précision sous-entend une durée de calcul plus importante.

La précision de l'analyse est paramétrable à travers les options d'EVA, notamment 2 options :

- `eva-auto-loop-unroll n`⁴ : cette option permet à EVA d'analyser les boucles (boucles `for`, boucles `while`), `n` représentant le nombre maximum d'itérations dans les boucles. Sans cette option, ou avec un `n` trop faible, il n'est pas possible à EVA de dérouler complètement les boucles présentes dans le code. Par conséquent, pour certaines itérations, l'interprétation réalisée par EVA ne sera pas suffisamment précise pour déterminer le comportement de la boucle et les valeurs prises par les variables dans la boucle.
- `eva-slevel n`⁵ : l'utilisation de cette option permet à EVA de garantir la précision de l'interprétation abstraite, en particulier pour les fonctions présentant des boucles ou des structures conditionnelles. Le `slevel` peut être vu comme un "carburant" donné à EVA, consommé en parcourant une fonction en présence de boucles et de structures conditionnelles⁶. En l'absence de "carburant" ou avec une quantité trop faible, EVA n'est pas capable de séparer l'ensemble des états possibles dans une fonction, et réalise une fusion de ces états. Par conséquent, il n'est pas possible à EVA de déterminer dans ce cas l'ensemble des valeurs possibles pour les différentes variables de cette fonction. Cette option, notamment quand le `slevel` est élevé, augmente fortement le temps de l'analyse. Pour réduire ce temps, tout en conservant la précision de l'analyse, il est possible d'utiliser l'option suivante `eva-slevel-function fcn :n` afin de personnaliser le `slevel` pour une fonction donnée. Il est ainsi possible d'augmenter le `slevel` pour les fonctions le nécessitant, et de prendre un `slevel` plus faible pour les autres fonctions⁷.

L'analyse des logs permet de vérifier la précision de l'analyse réalisée avec EVA. En effet, la présence du mot clé "merge" indique des imprécisions. Il est ainsi possible d'augmenter pour l'ensemble du code le `slevel` jusqu'à sa disparition, ou de ne l'augmenter que pour certaines fonctions.

options d'utilisation de FRAMAC

L'analyse avec EVA est réalisée en lançant la commande suivante :

```

1  frama-c usbctrl.c usbctrl_descriptors.c usbctrl_handlers.c
2  usbctrl_requests.c usbctrl_state.c framac/include/driver_api/usbotghs_frama.c \
3  -c11 -machdep x86_32 \
4  -no-frama-c-stdlib \
5  -warn-left-shift-negative \
6  -warn-right-shift-negative \
7  -warn-signed-downcast \
8  -warn-signed-overflow \
9  -warn-unsigned-downcast \
10 -warn-unsigned-overflow \
11 -kernel-msg-key pp \
12 -cpp-extra-args="-nostdinc -I framac/include" \
13 -rte \
14 -eva \
15 -eva-warn-undefined-pointer-comparison none \
16 -eva-auto-loop-unroll 500 \
17 -eva-slevel 500 \
18 -eva-slevel-function usbctrl_get_descriptor:12000 \
19 -eva-slevel-function usbotghs_send_data:1000 \
20 -eva-slevel-function usbotghs_endpoint_stall:1000 \
21 -eva-slevel-function usbotghs_endpoint_set_nak:1000 \
22 -eva-symbolic-locations-domain \
23 -eva-equality-domain \
24 -eva-bitwise-domain \

```

4. `n` entier positif

5. `n` entier positif

6. l'utilisation de cette fonction rend de fait optionnelle l'utilisation de `loop unroll`

7. le `slevel-function` est considéré à la place du `slevel` général

```

25 -eva-equality-through-calls-function usbctrl_start_device \
26 -eva-equality-through-calls-function usbctrl_is_valid_transition \
27 -wp-dynamic \
28 -eva-split-return auto \
29 -eva-partition-history 3 \
30 -eva-use-spec class_get_descriptor \
31 -eva-use-spec usbctrl_reset_received \
32 -eva-log a:frama-c-rte-eva.log \
33 -save framac/results/frama-c-rte-eva.session

```

Cette ligne de commandes est constituée :

- d'options pour le noyau de Frama-C (depuis -c11 jusqu'à -rte) :
 - des options nécessaires pour le parsing du code source, explicitées dans le paragraphe précédant
 - absolute-valid-range 0x40040000-0x400150000, afin d'indiquer à Frama-C que les adresses mémoires situées entre 0x40040000 et 0x400150000 sont valides. Cette plage d'adresse représente, pour le driver USB High Speed, les adresses mémoires du périphérique dans lesquelles le driver peut accéder en lecture et en écriture. Cette plage d'adresse est dépendante du driver USB utilisé et doit donc être modifiée en cas d'utilisation d'analyse d'un autre driver USB. Cette option est nécessaire car ces adresses ne sont pas allouées lors de l'exécution de la bibliothèque USBctrl, mais sont écrites dans un fichier source. Il fallait donc indiquer à Frama-C que ces adresses représentaient des adresses mémoires valides, accessibles en lecture et en écriture
 - des options afin de vérifier que le code ne contient pas d'overflow sur des entiers signés ou non signés, ou que des downcast d'entiers signés ou non signés ne sont pas réalisés par le code (les downcast et les overflow d'entiers sont des sources de RTE) (options préfixées par -warn)
 - l'option -rte permet à EVA et à WP d'annoter automatiquement le code avec des assertions ACSL, visualisables avec l'interface graphique (le code source n'est pas modifié). Cela permet de voir les assertions prouvées ou non par EVA, par exemple. La figure suivante illustre des assertions automatiques, prouvées et non prouvées par EVA :

FIGURE 4.5 – Visualisation des annotations automatiques avec l'interface graphique

Dans cet exemple...blablabla

- des options pour le greffon EVA, identifiées par le préfixe -eva :
 - eva-warn-undefined-pointer-comparison none : cette option permet de désactiver la vérification pour les comparaisons de pointeurs (très bof, à revoir...)
 - des options pour la précision de l'analyse : -eva-auto-loop-unroll, -eva-slevel et -eva-slevel-function (voir si plevel utile...)
 - eva-symbolic-locations-domain, -eva-bitwise-domain et -eva-equality-domain :
 - eva-split-return auto : pour avoir des états séparés à la fin de chaque fonction, afin d'avoir une analyse plus précise
 - eva-partition-history 3 : permet d'analyser avec précision les fonctions comportant de nombreuses structures conditionnelles (structures if/else, switch...)
 - eva-use-spec fcn : permet d'indiquer à EVA de ne prendre en compte que le contrat de la fonction fcn et non le corps de la fonction. Utile quand la fonction n'est pas définie, par exemple (ce qui est le cas pour les fonctions class_get_descriptor et usbctrl_reset_received).

Des options sont par ailleurs activées par défaut, en particulier l'option -wp-dynamic qui pour utiliser des annotations spécifiques pour les pointeurs de fonction. Cela permet à EVA d'analyser correctement les pointeurs de fonctions, en parcourant la fonction "pointée". Des annotations spécifiques sont toutefois nécessaires, comme le montre l'exemple suivant :

```

/*@ assert fcn &fcn1 ; */
/*@ calls fcn1 ; */

```

4.2.3 Utilisation de WP

La 3ème étape consiste à analyser le code avec WP, à la suite de l'analyse réalisée avec EVA. Comme expliqué dans le chapitre 2.5, cela permet à WP d'utiliser les résultats de l'analyse d'EVA afin notamment de prouver l'absence de RTE pour des cas plus complexes que ceux validés avec EVA.

stratégie d'analyse

Le greffon WP, contrairement au greffon EVA, ne nécessite pas de point d'entrée dans le code analysé. WP étant modulaire, l'analyse peut être réalisée fonction par fonction, après les avoir spécifiées avec un contrat de fonction écrit en ACSL (comme illustré dans le chapitre 2.4.3). L'objectif est d'obtenir pour l'ensemble des fonctions de la bibliothèques USBctrl une spécification validée par WP, comprenant des préconditions, des postconditions et des assigns, incluant le cas échéant des spécifications de boucles, afin de statuer sur l'absence de RTE et sur la bonne terminaison des fonctions.

La stratégie mise en place au cours de ce stage pour y parvenir a été la suivante :

- établir le contrat des fonctions de la bibliothèque USBctrl indépendantes des autres fonctions de la bibliothèque ou du driver USB
- établir le contrat des fonctions du driver USB appelées par la bibliothèque USBctrl
- établir le contrat des fonctions de la bibliothèque USBctrl qui appellent d'autres fonctions de la bibliothèque ou du driver USB.

Il est en effet nécessaire de commencer à spécifier les fonctions "appelées" avant de passer aux fonctions "appelantes" car le comportement des fonctions "appelantes" peut dépendre du résultat des fonctions "appelées".

Pour une fonction donnée, les spécifications se font en plusieurs étapes :

- établissement des préconditions (assumes et requires) et des postconditions, celles-ci portant sur le résultat de la fonction (par exemple, un code d'erreur). Il s'agit de spécifier la fonction sans rentrer finement dans le détail de son comportement, afin de vérifier son comportement général.
- spécification des boucles présentes dans les fonctions. Comme expliqué dans le chapitre 2.4.4, il s'agit, pour chaque boucle, de déterminer les propriétés valides avant d'entrer dans la boucle et qui le seront encore à la sortie de boucle (loop invariant), les variables modifiées par la boucle (loop assigns) et la propriété permettant de vérifier que la boucle se termine (loop variant)
- spécification des assigns dans les contrats de fonction, et ajout, le cas échéant, de post-conditions portant sur d'autres critères que la valeur de retour de la fonction en fonction de leur pertinence pour le comportement de la fonction. Par exemple, pour certaines fonctions modifiant l'état de la communication USB, il peut être pertinent de vérifier l'état atteint à la fin de la fonction. Cette dernière étape est la plus complexe, car elle nécessite d'une part d'identifier précisément les variables potentiellement modifiées par la fonction, mais également de comprendre quel était l'objectif recherché par la fonction, afin de déterminer des post-conditions représentatives de cet objectif.

options d'utilisation de FRAMAC

L'analyse avec WP est réalisée en lançant la commande suivante :

```
1  frama-c usbctrl.c usbctrl_descriptors.c usbctrl_handlers.c
2  usbctrl_requests.c usbctrl_state.c framac/include/driver_api/usbotghs_frama.c
3  -c11 -machdep x86_32 \
4  -absolute-valid-range 0x40040000-0x400150000 \
5  -no-frama-c-stdlib \
6  -warn-left-shift-negative \
7  -warn-right-shift-negative \
8  -warn-signed-downcast \
9  -warn-signed-overflow \
10 -warn-unsigned-downcast \
11 -warn-unsigned-overflow \
12 -kernel-msg-key pp \
```

```

13  -cpp-extra-args="-nostdinc -I framac/include" \
14  -rte \
15  -eva \
16  -eva-warn-undefined-pointer-comparison none \
17  -eva-auto-loop-unroll 500 \
18  -eva-slevel 500 \
19  -eva-slevel-function usbctrl_get_descriptor:12000 \
20  -eva-slevel-function usbbotghs_send_data:1000 \
21  -eva-slevel-function usbbotghs_endpoint_stall:1000 \
22  -eva-slevel-function usbbotghs_endpoint_set_nak:1000 \
23  -eva-symbolic-locations-domain \
24  -eva-equality-domain \
25  -eva-bitwise-domain \
26  -eva-equality-through-calls-function usbctrl_start_device \
27  -eva-equality-through-calls-function usbctrl_is_valid_transition \
28  -eva-split-return auto \
29  -eva-partition-history 10 \
30  -eva-use-spec class_get_descriptor \
31  -eva-use-spec usbctrl_reset_received \
32  -eva-log a:frama-c-rte-eva.log \
33  -then \
34  -wp \
35  -wp-model "Typed+ref+int" \
36  -wp-literals \
37  -wp-prover alt-ergo,cvc4,z3 \
38  -wp-timeout 15 \
39  -save $(SESSION) \
40  -time calcium_wp-eva.txt

```

Cette ligne de commande est constituée des options utilisées pour le parsing du code source et pour l'analyse avec EVA avec l'ajout des options suivantes :

- `-wp-model "Typed+ref+int"` : cette option indique le modèle mémoire utilisé dans l'analyse WP :
 - `typed` : dans ce modèle mémoire, chaque pointeur déclaré dans le code analysé est considéré avec un type qui ne varie pas. Par exemple, si une variable est déclarée en tant que (`uint_8*`), il ne sera pas possible de faire un cast vers un autre type de pointeur (WP ne sera pas capable dans ce cas de totalement valider les spécifications des fonctions dans lesquelles un tel cast serait fait)
 - `ref` :
 - `int` : permet d'utiliser des entiers de type machine (sur 32 bits par exemple) quand des overflows et ou des downcast peuvent se produire
- `-wp-literals` : permet d'exporter le contenu des chaînes de caractères aux prouveurs (sans cette option, blabla)
- `-wp-prover alt-ergo,cvc4,z3` : permet d'utiliser des prouveurs externes lors de l'analyse avec WP
- `-wp-timeout 15` : indique à WP que chaque prouveur externe dispose de 15s pour valider une obligation de preuve, avant celle-ci ne soit considérée comme non prouvée

Chapitre 5: Résultats obtenus

5.1 Résultats de l'analyse réalisée avec EVA

5.1.1 Travail réalisé

Comme indiqué dans le chapitre précédent, l'utilisation d'EVA nécessite un point d'entrée dans le code à analyser. La bibliothèque USBctrl ne disposant pas de fonction `main()`, j'ai défini plusieurs nouvelles fonctions spécifiques à l'analyse Frama-C, à partir desquelles l'ensemble des fonctions de la bibliothèque USBctrl et du driver USB High Speed sont appelées, avec différents contextes (afin de tester ces fonctions dans leur comportement nominal, c'est à dire avec des paramètres d'entrée corrects), et la gestion des erreurs de chacune de ces fonctions). Ces différentes fonctions sont ensuite appelées par une fonction principale, un `main()`, point d'entrée de l'analyse réalisée avec EVA. Ces fonctions ajoutées pour l'analyse avec EVA sont reproduites dans l'annexe 1.

Les fonctions introduites pour l'analyse avec EVA utilisent en particulier une fonction spécifique à EVA, la fonction `Frama_C_interval`, qui permet à EVA de parcourir les différentes fonctions analysées avec des paramètres d'entrées qui varient entre une valeur minimale et une valeur maximale. A titre d'illustration, il est possible de définir la variable suivante (un entier non signé sur 16 bits) :

```
uint16_t Value = Frama_C_interval(0,65535);
```

puis de l'utiliser dans une fonction. En appelant cette fonction une seule fois dans le point d'entrée de l'analyse, EVA analysera la fonction avec toutes les valeurs possibles de la variable entre 0 et 65535. La fonction `Frama_C_interval` est définie dans la bibliothèque C installée en même temps que Frama-C : cette bibliothèque, spécifique à l'utilisation de Frama-C, est une variante de la bibliothèque standard du C dans laquelle les différentes fonctions de la bibliothèque standard sont annotées en ACSL (ce qui permet de pouvoir utiliser si nécessaire les fonctions standards dans le code à analyser, sans avoir à d'abord définir leur contrat en ACSL). Par ailleurs, en plus des fonctions standards, la bibliothèque C de Frama-C introduit plusieurs fonctions spécifiques à l'analyse avec Frama-C, comme la fonction `Frama_C_interval`. Il existe plusieurs variante de cette fonction, en fonction du type de la variable que l'on souhaite faire varier. Dans le cadre de ce stage, seule la fonction `Frama_C_interval` a été utilisée. Il a toutefois été nécessaire d'inclure plusieurs fichiers de la bibliothèque C de Frama-C dans le répertoire `framac/include/` afin de pouvoir utiliser cette fonction.

Le taux de couverture d'EVA indiqué à la fin de l'analyse est de 91%, l'ensemble des fonctions de la bibliothèque USBctrl et du driver USB High Speed étant analysé à l'aide d'EVA. Parmi les embranchements non analysés par EVA se trouvent :

- des embranchements non atteignables, tels que :
 - des case default non atteignable car une structure conditionnelle avant le switch élimine ce cas, mais ajoutés à des fins de compilation
 - des structures conditionnelles redondantes
 - des cas d'erreur non atteints (pointeurs null par exemple, ou paramètres invalides dans certaines structures)

Pour ces embranchements non analysés avec EVA, je les ai analysés à l'aide de l'interface graphique, afin de déterminer si une RTE était possible. Je n'en ai pas trouvé dans ces embranchements.

Les options présentées dans le chapitre précédent garantissent par ailleurs la précision de l'analyse, le temps de calcul étant de près de XX minute pour un processeur de ¹

5.1.2 RTE découverts

liste sous forme de tableau ? integer overflow, division par 0, bad memory access, variables non initialisées, tests manquants

Ces RTE ont été patchés, détail en annexe ?

Type de RTE	fonctions impactées	Patch
débordement d'entier non signé	test	test
downcast d'entier non signé	test	test
débordement de tableaux	test	test
accès mémoire invalide	test	test
division par 0	test	test
variables non initialisées	test	test

5.1.3 Problèmes rencontrés

établir un point d'entrée correct : déterminer, parmi l'ensemble des fonctions, les données d'entrées nécessaires aux fonctions. comment atteindre tout le code source ? nécessité d'initialiser des structures incorrectes, avec des valeurs incorrectes (par exemple pour les pointeurs null). Jouer sur la machine à état : certains états ne sont possibles qu'après interaction avec la machine hôte par exemple. Solution : frama-c_interval sur une structure globale, avant l'appel à la fonction

boucles : unroll : quel niveau ? examen des boucles, pas besoin de faire bcp (donc très bonne approximation) certaines fonctions comportent de nombreuses boucles conditionnelles imbriquées : utilisation de l'option domain history nécessaire pour qu'EVA puisse déterminer les valeurs possibles des variables dans les différentes conditions du code : plus généralement, la précision de l'analyse

pointeurs de fonction : utilisation de calls + wp dynamic : permet à EVA de savoir vers quoi pointe le pointeur de fonction. Nécessité de rajouter des assertions dans le code. Important pour WP : wp pourra prouver la fonction grâce aux résultats d'EVA (sans ça, WP n'est pas capable de prouver la partie du code dans laquelle se trouve le pointeur de fonction)

5.1.4 Pistes d'amélioration

5.2 Résultats obtenus avec WP

5.2.1 Travail réalisé

chiffrer à la fin le pourcentage de fonctions spécifiées dans la lib USB s'il en manque, ça sera expliqué dans problèmes rencontrés

5.2.2 Problèmes rencontrés

aliasing : faire un define backend pour que wp passe les preuves volatile : problème, aucune preuve ne passe si une variable est volatile (car elle peut prendre n'importe quelle valeur : donc retrait des volatile) memset / memcpy : initialisation des structures "manuellement" et copie manuelle également pointeurs de fonction : wp dynamic + calls + ajout d'assertion. Si c'est validé par EVA, WP le prend pour argent comptant, et la preuve passe boucle while : difficulté à déterminer des variant pour ces boucles (quand while (f(a)) par exemple, donc transformation de ces boucles while en boucles for avec ajout d'un compteur suffisamment grand, dépendant du while. Mettre deux exemples de compteur pour illustrer

de manière générale, difficulté à spécifier certaines boucles : exemples de boucles avec un if où on sort si on rentre dedans. Si on sort, c'est une conditions anormale de sortie de boucle, donc il ne faut pas en tenir compte dans les invariant et les assigns de la boucle (subtil...)

1. il n'est actuellement pas possible de paralléliser l'analyse d'EVA, donc le nombre de coeur n'influe pas sur le temps de calcul

réponse : utiliser des variables ghost (type de variable ACSL) pour "remplacer les variables statiques" dans les spécifications des fonctions. Concrètement, on assign la variable ghost à la variable static quand celle-ci est modifiée par une fonction (et donc apparait dans les assign de la spec de la fonction). Mais c'est juste un artifice pour compenser le fait que frama ne gère pas correctement les variables statiques

5.2.3 Résultats obtenus

donner combien de fonctions sont pleinement spécifiées combien de problème il reste (s'il en reste...)

5.3 Retour d'expérience sur l'utilisation de Frama-C

complexe mais puissante importance des options : pas facile de trouver des explications (toutes ne sont pas expliquées dans les différents manuels, frama -wp-h pour avoir la liste exhaustive, mais explications succinctes

- installation pas facile également !

- pas facile d'obtenir des renseignements en ligne pour comprendre certains comportements de framaC (cf chapitre plus loin) : contact du CEA nécessaire

- importance de la communauté en ligne

Toutefois, les exemples donnés ne sont pas toujours bien illustrés, et toutes les options ne sont pas présentes. Il est ainsi nécessaire de se référer à l'aide en ligne de commande, installée en même temps que Frama-C.

Conclusion

...

Annexe A: Point d'entrée dans la bibliothèque USBctrl

3 fonctions ont été codées afin de pouvoir analyser la bibliothèque USBctrl et le driver USB High Speed avec EVA :

- `test_fcn_usbctrl` : analyse des fonctions de bibliothèque USBctrl dans leur comportement nominal :
 - initialisation des paramètres d'entrée des différentes fonctions de la bibliothèques USBctrl à l'aide de la fonction de Framac C `Frama_C_interval`,
 - déclaration de deux contextes USB, de plusieurs interfaces USB,
 - appel des fonctions de gestion des événements et des requêtes.
- `test_fcn_usbctrl_erreur` : vérification de la gestion d'erreurs implémentée dans la bibliothèque USBctrl, avec des paramètres d'entrée invalides (pointeurs null par exemple, ou dépassant une valeur maximale...),
- `test_fcn_driver_eva` : analyse de la gestion d'erreur des fonctions du driver et analyse des fonctions du driver USB High Speed qui ne sont pas directement appelées par la bibliothèque USBctrl et test

```
1 void test_fcn_usbctrl() {
2
3     uint32_t dev_id = Framac_C_interval(0,65535) ;
4     uint32_t size = Framac_C_interval(0,65535) ;
5     uint32_t handler ;
6     uint8_t ep = Framac_C_interval(0,255);
7     uint8_t iface = Framac_C_interval(0,MAX_INTERFACES_PER_DEVICE-1);
8     uint8_t ep_number = Framac_C_interval(0,MAX_EP_PER_INTERFACE);
9     uint8_t EP_type = Framac_C_interval(0,3);
10    uint8_t EP_dir = Framac_C_interval(0,1);
11    uint8_t USB_class = Framac_C_interval(0,17);
12    uint32_t USBdc_i_handler = Framac_C_interval(0,65535) ;
13    usb_device_trans_t transition = Framac_C_interval(0,MAX_TRANSITION_STATE-1) ;
14    usb_device_state_t current_state = Framac_C_interval(0,9);
15    usbctrl_request_code_t request = Framac_C_interval(0x0,0xc);
16
17    uint8_t RequestType = Framac_C_interval(0,255);
18    uint8_t Request = Framac_C_interval(0,0xd);
19    uint16_t Value = Framac_C_interval(0,65535);
20    uint16_t Index = Framac_C_interval(0,65535);
21    uint16_t Length = Framac_C_interval(0,65535);
22
23    usbctrl_interface_t iface_1 =
24    { .usb_class = USB_class, .usb_ep_number = ep_number, .dedicated = true,
25    .eps[0].type = EP_type, .eps[0].dir = EP_dir, .eps[0].handler = handler_ep,
26    .rqst_handler = class_rqst_handler, .class_desc_handler = class_get_descriptor};
27
28    usbctrl_interface_t iface_2 =
29    { .usb_class = USB_class, .usb_ep_number = ep_number, .dedicated = true,
30    .eps[0].type = EP_type, .eps[0].dir = EP_dir, .eps[0].handler = handler_ep,
31    .rqst_handler = class_rqst_handler, .class_desc_handler = class_get_descriptor};
32
33    usbctrl_interface_t iface_3 =
34    { .usb_class = USB_class, .usb_ep_number = ep_number, .dedicated = false,
```

```

35 .eps[0].type = EP_type, .eps[0].dir = EP_dir, .eps[0].handler = handler_ep});
36
37 usbctrl_setup_pkt_t pkt =
38 { .bmRequestType = RequestType, .bRequest = Request, .wValue = Value,
39   .wIndex = Index, .wLength = Length };
40
41
42 usbctrl_context_t *ctx1 = NULL;
43 usbctrl_context_t *ctx2 = NULL;
44 uint32_t ctxh1=0;
45 uint32_t ctxh2=0;
46
47 ////////////////////////////////////////////////////
48 //          premier context
49 ////////////////////////////////////////////////////
50
51 usbctrl_declare(6, &ctxh1);
52 usbctrl_initialize(ctxh1);
53 usbctrl_get_context(6, &ctx1);
54 usbctrl_declare_interface(ctxh1, &iface_1) ;
55 usbctrl_declare_interface(ctxh1, &iface_2);
56 usbctrl_declare_interface(ctxh1, &iface_3);
57 usbctrl_get_interface(ctx1, iface);
58 usbctrl_get_handler(ctx1, &handler);
59 usbctrl_is_interface_exists(ctx1, iface);
60 usbctrl_is_endpoint_exists(ctx1, ep);
61 usbctrl_start_device(ctxh1) ;
62 usbctrl_stop_device(ctxh1) ;
63
64 if(ctx1 != NULL){
65     ctx1->state = Frama_C_interval(0,9);
66     usbctrl_is_valid_transition(ctx1->state, transition, ctx1);
67     usbctrl_handle_class_requests(&pkt, ctx1) ;
68 }
69
70 ////////////////////////////////////////////////////
71 //          2nd context
72 ////////////////////////////////////////////////////
73
74 usbctrl_declare(7, &ctxh2);
75 usbctrl_initialize(ctxh2);
76 usbctrl_get_handler(&ctx_test, &handler);
77 usbctrl_get_context(7, &ctx2);
78 usbctrl_get_handler(ctx2, &handler);
79 usbctrl_declare_interface(ctxh2, &iface_1) ;
80 usbctrl_declare_interface(ctxh2, &iface_2);
81 usbctrl_declare_interface(ctxh2, &iface_3);
82 usbctrl_get_interface(ctx2, iface);
83 usbctrl_is_interface_exists(ctx2, iface);
84 usbctrl_is_endpoint_exists(ctx2, ep);
85 usbctrl_start_device(ctxh2) ;
86 usb_device_state_t state = usbctrl_get_state(ctx2);
87 usbctrl_stop_device(ctxh2) ;
88
89 if(ctx2 != NULL){
90     ctx2->state = Frama_C_interval(0,9);
91     usbctrl_is_valid_transition(ctx2->state, transition, ctx2);
92     usbctrl_handle_class_requests(&pkt, ctx2) ;
93 }
94
95 ////////////////////////////////////////////////////
96 //          fonctions qui vont utiliser les deux contextes (inepevent et outepevent)
97 ////////////////////////////////////////////////////
98
99 ctx_list[0].ctrl_req_processing = true;
100 usbctrl_handle_inepevent(dev_id, size, ep);
101 usbctrl_handle_outepevent(dev_id, size, ep);
102 usbctrl_handle_earlysuspend(dev_id) ;
103 usbctrl_handle_usbsuspend(dev_id);

```

```

105     usbctrl_handle_wakeup(dev_id) ;
106     usbctrl_std_req_get_dir(&pkt) ;
107     usbctrl_handle_reset(dev_id);
108     usbctrl_next_state(current_state, request);
109     usbctrl_handle_requests(&pkt, dev_id) ;
110     usbctrl_handle_requests_switch(&pkt, dev_id) ;
111
112 }

1 void test_fcn_usbctrl_erreur() {
2
3     uint32_t dev_id = Frama_C_interval(0, RAND_UINT_32-1) ;
4     uint32_t size = Frama_C_interval(0, RAND_UINT_32-1) ;
5     uint32_t ctxh = Frama_C_interval(0, MAX_USB_CTRL_CTX-1);
6     uint32_t handler = Frama_C_interval(0, RAND_UINT_32-1);
7     uint8_t ep = Frama_C_interval(0, 255);
8     uint8_t iface = Frama_C_interval(0, MAX_INTERFACES_PER_DEVICE-1);
9     uint8_t ep_number = Frama_C_interval(0, MAX_EP_PER_INTERFACE);
10    uint8_t EP_type = Frama_C_interval(0, 3);
11    uint8_t EP_dir = Frama_C_interval(0, 1);
12    uint8_t USB_class = Frama_C_interval(0, 17);
13    uint32_t USBdci_handler = Frama_C_interval(0, RAND_UINT_32-1) ;
14
15
16    uint8_t RequestType = Frama_C_interval(0, 255);
17    uint8_t Request = Frama_C_interval(0, 255);
18    uint16_t Value = Frama_C_interval(0, 65535);
19    uint16_t Index = Frama_C_interval(0, 65535);
20    uint16_t Length = Frama_C_interval(0, 65535);
21
22    usbctrl_setup_pkt_t pkt =
23    { .bmRequestType = RequestType, .bRequest = Request, .wValue = Value,
24      .wIndex = Index, .wLength = Length };
25    usbctrl_interface_t iface_1 =
26    { .usb_class = USB_class, .usb_ep_number = ep_number, .dedicated = true,
27      .eps[0].type = EP_type, .eps[0].dir = EP_dir, .eps[0].handler = NULL,
28      .rqst_handler = NULL, .class_desc_handler = NULL};
29    usbctrl_interface_t iface_2 =
30    { .usb_class = USB_class, .usb_ep_number = ep_number, .dedicated = true,
31      .eps[0].type = EP_type, .eps[0].dir = EP_dir, .eps[0].handler = NULL,
32      .rqst_handler = NULL, .class_desc_handler = NULL};
33
34    usbctrl_context_t ctx_test = { .dev_id = 8, .address= 2};
35
36    /*
37     usbctrl_declare : cas d'erreur - pointeur ctxh null
38                      - num_ctx >= 2
39    */
40
41    uint32_t *bad_ctxh = NULL;
42    usbctrl_declare(dev_id, bad_ctxh);
43
44    ctxh = 1 ;
45    num_ctx = 2;
46    //@ ghost GHOST_num_ctx = num_ctx ;
47    usbctrl_declare(dev_id, &ctxh);
48
49
50
51    /*
52     usbctrl_declare : cas d'erreur : ctxh >= num_ctx
53    */
54
55    ctxh = 0 ;
56    num_ctx = 1 ;
57    usbctrl_initialize(ctxh);
58
59
60    ctxh = 1 ;
61    num_ctx = 0 ;
62    //@ ghost GHOST_num_ctx = num_ctx ;

```

```

63     usbctrl_declare(8, &ctxh);
64     usbctrl_initialize(ctxh);
65
66     /*
67         usbctrl_declare_interface : cas d'erreur - ctxh >= num_ctx
68                                     - pointeur iface == null
69                                     - interface_num >=
70
71     MAX_INTERFACES_PER_DEVICE
72                                     - pkt_maxsize > usbotghs_get_ep_mpsize()
73     Dans le cas nominal, avec le test sur 2 interfaces, num_cfg >= MAX_USB_CTRL_CTX-1
74     donc une partie du code n'est pas atteinte. Cas traite ci-dessous, quand on
75     rajoute une interface de controle
76 */
77
78     ctxh = 2 ;
79     num_ctx = 1 ;
80     //@ ghost GHOST_num_ctx = num_ctx ;
81     usbctrl_declare_interface(ctxh, &iface_1) ;
82
83     ctxh = 0 ;
84     num_ctx = 1 ;
85     usbctrl_interface_t *iface_null = NULL ;
86     usbctrl_declare_interface(ctxh, iface_null) ;
87
88     usbctrl_interface_t iface_3 =
89     { .usb_class = 0, .usb_ep_number = 2, .dedicated = true, .eps[0].type = 3,
90       .eps[0].pkt_maxsize = MAX_EPx_PKT_SIZE + 1 };
91     ctx_list[ctxh].cfg[0].interface_num = MAX_INTERFACES_PER_DEVICE ;
92     usbctrl_declare_interface(ctxh, &iface_3) ;
93
94     usbctrl_interface_t iface_4 =
95     { .usb_class = 0, .usb_ep_number = 2, .dedicated = false, .eps[0].type = 3,
96       .eps[0].pkt_maxsize = MAX_EPx_PKT_SIZE + 1 };
97     ctx_list[ctxh].cfg[0].interface_num = MAX_INTERFACES_PER_DEVICE - 1 ;
98     //ctx_list[ctxh].cfg[0].interfaces[0].eps[0].pkt_maxsize = MAX_EPx_PKT_SIZE + 1 ;
99     usbctrl_declare_interface(ctxh, &iface_4) ;
100
101     //ctx_list[ctxh].cfg[0].interface_num = MAX_INTERFACES_PER_DEVICE - 1 ;
102     //ctx_list[ctxh].num_cfg < MAX_USB_CTRL_CTX - 1 ;
103     //usbctrl_declare_interface(ctxh, &iface_3) ;
104
105     /*
106         usbctrl_get_interface : cas d'erreur - pointeur ctx null
107         cas iface < ctx->cfg[ctx->curr_cfg].interface_num pas atteint dans le cas nominal
108     */
109     usbctrl_context_t *bad_ctx = NULL ;
110     usbctrl_get_interface(bad_ctx, iface);
111
112     ctx_list[ctxh].cfg[0].interface_num = MAX_INTERFACES_PER_DEVICE ;
113     usbctrl_get_interface((usbctrl_context_t *)&(ctx_list[ctxh]), iface);
114
115     /*
116         usbctrl_get_handler: cas d'erreur - pointeur ctx null
117                                     - pointeur handler null
118                                     - context different de ctx_list,
119                                     pour trigger certains cas dans get_handler
120     comme num_ctx < MAX_USB_CTRL_CTX pour ne pas avoir de debordement de tableau
121     la boucle n'est parcourue qu'une fois dans la fonction
122     */
123     usbctrl_get_handler(bad_ctx, &handler);
124     usbctrl_get_handler(&ctx_test, &handler); // pour tester behavior not_found
125
126
127     /*
128         usbctrl_get_context, usbctrl_is_endpoint_exists && usbctrl_is_interface_exists:
129         cas d'erreur - pointeur ctx null
130     */
131

```



```

132     usbctrl_get_context(dev_id, NULL);
133     usbctrl_is_endpoint_exists(bad_ctx, ep);
134     usbctrl_is_interface_exists(bad_ctx, iface);
135
136     /*
137      * test erreur avec un numero de ctx >= num_ctx (qui vaut 1 au max dans mon cas,
138      * avec un max de cfg de 2)
139      */
140
141     usbctrl_start_device(4) ;
142     usbctrl_stop_device(4) ;
143
144     /*
145      * test erreur sur get_descriptor : parcourir tous les types possibles,
146      * incluant un faux type
147      * pointeurs null
148      * ctx != ctx_list[i] pour error_not_found dans get_handler
149      * class_get_descriptor : error_none forcement, donc je ne rentre pas dans
150      * errcode != error_none
151      */
152
153     uint8_t buf[255] = {0} ;
154     uint32_t desc_size = 0 ;
155     usbctrl_context_t ctx1 = {1} ;
156
157     usbctrl_get_descriptor(9,&buf[0],&desc_size,&ctx1, &pkt);
158     usbctrl_get_descriptor(USB_DESC_DEV_QUALIFIER,&buf[0],&desc_size,&ctx1, &pkt);
159     usbctrl_get_descriptor(USB_DESC_OTHER_SPEED_CFG,&buf[0],&desc_size,&ctx1, &pkt);
160     usbctrl_get_descriptor(USB_DESC_IFACE_POWER,&buf[0],&desc_size,&ctx1, &pkt);
161     usbctrl_get_descriptor(1,NULL,&desc_size,&ctx1, &pkt);
162     usbctrl_get_descriptor(1,&buf[0],NULL,&ctx1, &pkt);
163     usbctrl_get_descriptor(1,&buf[0],&desc_size,NULL, &pkt);
164     usbctrl_get_descriptor(1,&buf[0],&desc_size,&ctx1, NULL);
165
166     usbctrl_get_state(NULL) ;
167     usbctrl_set_state(&ctx1,10);
168     usbctrl_set_state(NULL,10);
169
170     usbctrl_context_t ctx2 = ctx_list[0] ;
171     ctx2.state = Frama_C_interval(0,9);
172     usbctrl_handle_class_requests(&pkt, &ctx2);
173
174     usbctrl_handle_requests(NULL, dev_id);
175 }

```

```

1 void test_fcn_driver_eva(){
2
3     uint8_t ep_id = Frama_C_interval(0,255);
4     uint8_t ep_num = Frama_C_interval(0,255);
5     uint8_t dir8 = Frama_C_interval(0,255);
6     uint8_t dst = Frama_C_interval(0,255);
7     uint32_t size = Frama_C_interval(0,65534);
8     uint8_t fifo = Frama_C_interval(0,255);
9     uint32_t fifo_idx = Frama_C_interval(0,65535);
10    uint32_t fifo_size = Frama_C_interval(0,65535);
11
12    uint8_t src = 1 ;
13
14    usbotghs_ep_dir_t dir = Frama_C_interval(0,1);
15    usbotghs_ep_type_t type = Frama_C_interval(0,3);
16    usbotghs_ep_state_t state = Frama_C_interval(0,9) ;
17
18    usbotghs_global_stall() ;
19    usbotghs_endpoint_set_nak(ep_id, dir) ;
20    usbotghs_global_stall_clear();
21    usbotghs_endpoint_stall_clear(ep_id, dir);
22    usbotghs_deconfigure_endpoint(ep_id);
23    usbotghs_activate_endpoint(dir8, dir);
24    usbotghs_deactivate_endpoint(ep_id, dir);
25    usbotghs_endpoint_nak(ep_id);
26    usbotghs_endpoint_nak_clear(ep_id);

```

```
27     usbotghs_endpoint_disable( ep_id, dir);
28     usbotghs_endpoint_enable( ep_id, dir);
29     usbotghs_endpoint_clear_nak(ep_id, dir) ;
30     usbotghs_endpoint_stall(ep_id, dir) ;
31     usbotghs_get_ep_state(ep_id, dir) ;
32
33     uint8_t resp[1024] = { 0 };
34     usbotghs_send_zlp(ep_id);
35     usbotghs_txfifo_flush(ep_id);
36     usb_backend_drv_configure_endpoint(ep_id, type, dir, size, USB_BACKEND_EP_ODDFRAME, &
        handler_ep);
37 }
```