C2TLA+ : Traduction automatique du code C vers TLA+

Amira Methni, Matthieu Lemerre, Belgacem Ben Hedia, Kamel Barkaoui, Serge Haddad

C2TLA+ : Traduction automatique du code C vers TLA+

Amira Methni, Matthieu Lemerre, Belgacem Ben Hedia
CEA, LIST, Point Courrier 94
91191 Gif-sur-Yvette, France
Email : nom.prénom@cea.fr

Kamel Barkaoui
CEDRIC Laboratory, CNAM
292 rue St-Martin
75141 Paris, France
Email : kamel.barkaoui@cnam.fr

Serge Haddad
LSV, CNRS & ENS de Cachan
61, avenue du Président Wilson
94235 CACHAN Cedex, France
Email : haddad@lsv.ens-cachan.fr

***Résumé*—Nous nous intéressons dans ce papier à l'automatisation de la traduction d'un code source C vers un modèle écrit dans le langage de spécification TLA+. Nous proposons alors un outil C2TLA+ pour automatiser le passage d'un code source C vers un modèle écrit dans un langage combinant une logique temporelle avec une logique des actions afin qu'il soit vérifié par le model-cheker TLC. Ce papier illustre les règles de représentation et de traduction utilisées pour passer d'une implémentation à une spécification TLA+.**

I. Introduction

Suite au progrès technologique, la taille des logiciels et leurs complexités n'ont pas cessé de s'accroître. Néanmoins, ces systèmes peuvent comporter des dysfonctionnements ou des erreurs de conception ou d'implémentation, dont les conséquences peuvent être catastrophiques. De tels systèmes sont dits critiques. Leur développement fait alors appel à plusieurs méthodes et techniques dont l'objectif principal est d'assurer la sûreté de fonctionnement de ces systèmes. D'où l'intérêt des méthodes formelles, qui se basent sur des notions mathématiques et logiques et reposent sur une sémantique précise [5]. Ces méthodes permettent de vérifier de manière rigoureuse le bon fonctionnement des logiciels et plus généralement des systèmes. Pour ce faire, il est nécessaire de modéliser et vérifier le comportement du système et les propriétés qu'il doit satisfaire en utilisant un langage formel.
Plusieurs méthodes formelles existent. Le *model-checking* constitue une des méthodes les plus répandues. Il regroupe plusieurs techniques automatiques avec lesquelles les propriétés sont vérifiées en explorant tous les états possibles du système. Formellement, étant donné un modèle M du système à vérifier et une propriété (formule) φ, le *model-checking* [15] consiste à répondre à la question : $M \models \varphi$, c'est à dire on cherche à confirmer si M satisfait φ.
Nous nous intéressons dans notre étude aux systèmes temps réels bas niveau, particulièrement les noyaux des systèmes d'exploitation écrits en C. Nous utilisons TLA+ [14] comme langage de spécification de ces systèmes. Le choix s'est porté sur TLA+ pour plusieurs raisons. En effet, sa puissance d'expression basée sur la théorie des ensembles permet de spécifier des systèmes parallèles. TLA+ permet de décrire le comportement d'un système et ses propriétés à la fois dans un seul formalisme logique et dispose d'outils : des analyseurs syntaxiques et sémantiques, des outils de preuve et un *model-checker*.
Notre objectif est de traduire un programme C en une spécification équivalente écrite en TLA+ afin de vérifier ses propriétés. Dans ce papier, nous décrivons le processus de traduction du code source C en TLA+.
Ce papier est organisé de la manière suivante : la Section 2 présente un aperçu sur les travaux existants dans la vérification des programmes C. La Section 3 introduit le langage TLA+ et son *model-checker* TLC. Dans la Section 4, nous décrivons les règles de traduction d'un sous-ensemble de C vers TLA+ avec des exemples. Nous illustrons par la suite la technique d'implémentation avec une étude de cas.

II. Travaux existants

Modex (*Model Extractor*) [1] est une extension de l'outil AX [9] et permet l'extraction automatique d'un modèle à partir du code source C. Le modèle de sortie est décrit en Promela afin qu'il soit vérifié avec l'outil SPIN [8]. Certains aspects de cet outil on été repris dans les travaux de [11].
Des travaux basés sur l'analyse déductive se sont intéressés à la vérification des programmes C. Nous citons la plate-forme Frama-C [3], qui propose plusieurs plugins destinés à faire des tâches spécifiques d'analyse statique sur un programme C. D'autres travaux dans le même contexte se sont intéressés à la vérification des programmes Java, comme JavaPathFinder [6]. Des outils comme BLAST [7] et SLAM [4] combinent plusieurs techniques d'analyse statique et de *model-checking* pour la vérification des programmes C.
TLA+ fournit un cadre pour la spécification et la vérification des propriétés sur le système. Ce travail constitue une phase permettant de rendre une implémentation C vérifiable directement par TLC. Il s'agit du premier travail qui propose un processus automatique de traduction d'un programme C en une spécification TLA+.

III. Langage TLA+

TLA+ (*Temporal Logic of Actions*) [14] est un langage formel de spécification développé par Leslie Lamport pour décrire les systèmes réactifs. TLA+ est basé sur la logique TLA [13] pour vérifier des propriétés spécifiées sur un système. La sémantique de la logique TLA [13] est fondée sur les suites d'états ou les traces. Un état s est une valuation de

variables, il s'agit d'une fonction de *Var* vers *Val* : $s \in [Var \rightarrow Val\ ]$. Une suite d'état ou trace est une suite infinie d'états. Une suite d'états est un comportement. La notion d'action permet de raisonner sur un couple d'états. Une action est une expression booléenne contenant des variables primées et /ou non primées et représente une relation entre deux états : un ancien état et un nouvel état. Les variables non primées appartiennent à l'ancien état et les variables primées appartiennent au nouvel état. Une action spécifie alors l'ensemble de transitions d'états autorisées. Ainsi, une spécification TLA+ simule tous les comportements autorisés lors de l'exécution du système. Les spécifications TLA+ sont partitionnées en *modules*.

Pour vérifier des propriétés sur un système, TLA+ dispose d'un *model-checker* TLC (*Temporal Logic Checker*) [17]. Il permet de vérifier automatiquement si les spécifications écrites en TLA+ satisfont les propriétés temporelles définies en générant les différents états possibles du système. TLC prend en entrée un module TLA+ et un fichier de configuration définissant les propriétés à vérifier. Les spécifications gérées par TLC s'écrivent sous la forme suivante :

$$Spec \triangleq Init \wedge \Box[Next]_{vars} \wedge Temporal$$

Cette spécification est composée d'un prédicat d'état $Init$ qui définit l'état initial du système, autrement dit les valeurs initiales des variables utilisées. Ensuite, une action appelée $Next$ qui définit les transitions possibles du système, autrement dit comment les variables changent de valeurs à l'état suivant. Et une conjonction de l'ensemble des formules temporelles $Temporal$, que le système doit satisfaire.

IV. Traduction C en TLA+

Dans ce travail, nous considérons qu'un programme C est formé d'une *liste de déclarations* et une *liste de définitions de fonctions*. La syntaxe et la sémantique de ANSI-C sont données dans [2]. Dans ce papier nous nous limitons à un sous-ensemble de celui-ci et nous ne traitons pas les pointeurs et les fonctions récursives. La définition d'une fonction, qu'on appelle corps de la fonction est formé d'une *liste de déclarations de variables locales* et une *liste d'instructions*. Dans TLA+, un module est la plus petite unité de structure. Il est composé des parties suivantes : une déclaration des paramètres, des définitions de prédicats et des théorèmes spécifiant des propriétés à vérifier par le module.

Pour faciliter l'écriture des formules TLA+, Lamport a proposé un format de représentation des formules dans [12] utilisant des connecteurs $\wedge$ et $\vee$ (désignant respectivement le *ET* et le *OU* logique) sous forme préfixée. Dans cette section, nous illustrons les règles de traduction d'un sous-ensemble du langage C.

A. Déclaration des variables

TLA+ est un langage non typé, on n'a pas alors à spécifier le type de variable à déclarer. En outre, la notion de variable locale n'existe pas dans TLA+. Toutes les variables globales et locales définies dans C sont déclarées dans TLA+ avec le mot-clé `VARIABLES`. Les variables locales sont traduites en spécifiant dans leurs identificateurs le nom de la fonction dans laquelle elles sont déclarées. Ceci permettra d'enlever l'ambiguïté dans le cas où plusieurs fonctions déclareraient des variables locales avec les mêmes noms d'une part et permet une lisibilité au niveau des traces produites par le *model-checking* d'autre part. L'exemple de la FIGURE 1 présente la

```c
int x = 0;
void fonction1()
{ int i;
  /* Corps de la fonction1 */ }
void fonction2()
{ int i;
  /* Corps de la fonction2 */ }
void main()
{ /* Corps de la fonction
  main  */ }
```

(a) Code C

```
------- MODULE module --------
EXTENDS Naturals, TLC
VARIABLES x,fonction1_i,fonction2_i

vars == << x,fonction1_i,fonction2_i >>

(* Corps du module *)
==============================
```

(b) Code TLA+

FIGURE 1: Traduction des déclarations de variables

traduction d'un code C définissant une variable globale `x`, deux définitions de fonctions avec chacune une déclaration d'une variable locale `i`. Le code traduit déclare la variable globale `x`, et renomme les deux variables locales en `fonction1_i` et `fonction2_i` respectivement pour les fonctions `fonction1` et `fonction2`.

B. Les instructions

Un programme C est une série d'instructions effectuées l'une à la suite de l'autre, de la première à la dernière, de manière séquentielle. Par opposition à TLA+ qui est plutôt basée sur une logique utilisant des opérateurs et des prédicats, autrement dit des formules logiques qui peuvent prendre la valeur `vraie` ou `faux` à un état déterminé. Pour exprimer le séquencement d'un programme C, nous définissons dans TLA+ des étiquettes. Une étiquette est une variable, qui sert comme identificateur désignant l'état courant du système, autrement dit l'instruction C à exécuter. Dans notre cas, nous supposons que les valeurs des étiquettes sont de type chaîne de caractères.

L'insertion des étiquettes doit obéir à certaines règles :

- Chaque instruction C reçoit une étiquette ou `label`. Nous identifions l'instruction TLA+ par cette étiquette unique.
- La traduction de l'instruction `return` est une action dont l'étiquette a la valeur `"done"`.

```c
/* Liste d'instructions */

  x = x + 1;
  y = x + y;
  x = y*2;
  ...
```

(a) Code C

```
\/
  /\ main_label = "lbl_3"
  /\ x' = x + 1
  /\ main_label' = "lbl_4"
  /\ UNCHANGED <<y>>
\/
  /\ main_label = "lbl_4"
  /\ y' = x + y
  /\ main_label' = "lbl_5"
  /\ UNCHANGED <<x>>
\/
  /\ main_label = "lbl_5"
  /\ x' = y * 2
  /\ main_label' = "lbl_6"
  /\ UNCHANGED <<y>>
  ...
```

(b) Code TLA+

FIGURE 2: Traduction d'une suite d'instructions

La FIGURE 2 illustre un exemple de traduction d'une suite d'instructions C simples, sans modification du flot de contrôle. Une instruction C est traduite en une conjonction de formules :

- Une formule définissant l'étiquette de l'instruction courante,

– Une formule définissant l'opération de l'instruction en question (affectation, instruction conditionnelle, etc.),
– Une formule définissant les valeurs des variables à l'état suivant (présentées par des variables primées), notamment l'étiquette de la prochaine action à évaluer. Si la valeur de la variable est inchangée, elle est alors spécifiée par le mot-clé `UNCHANGED`.

Le traducteur introduit dans le module TLA+ une nouvelle variable nommée `nom_fonction_label` dans la liste de déclarations de variables, avec `nom_fonction` spécifie le nom de la fonction où l'instruction est définie.

C. Les fonctions

La syntaxe et la sémantique de TLA+ sont différentes de celles de C. En effet, il n'existe pas de notion de fonction dans TLA+, il est donc nécessaire de faire l'émulation de celles-ci. Le concept de TLA+ se base sur les prédicats et les formules logiques, que nous utilisons pour émuler une fonction. Chaque définition de fonction est donc traduite en un prédicat dans TLA+ qui porte le même nom que la fonction C. Pour chaque fonction, nous ajoutons la variable qui contrôle les étapes d'exécution de la fonction comme décrite dans la section IV-B.

1) Définition de fonction: Le corps de la fonction C est un *bloc* contenant une *liste de déclarations* et une *liste instructions*. La traduction du corps d'une fonction revient à la traduction d'un bloc d'instructions comme décrit dans la section IV-B.

2) L'appel de fonction: La traduction d'un appel de fonction est basée sur le même principe que les autres types d'instructions, mais en ajoutant des conditions pour gérer la synchronisation entre la fonction appelante et la fonction appelée. Un retour de fonction est identifié par une étiquette spécifique qui prend la valeur `"done"`. Cette représentation permet de contrôler les étapes d'exécution de la fonction appelante. Nous spécifions alors une formule qui définit si l'étiquette de l'état actuel n'est pas égale à `"done"` alors nous évaluons le prédicat de la fonction. Sinon, nous évaluons l'action suivante du prédicat de la fonction appelante et nous réinitialisons l'étiquette de la fonction appelée à la valeur de l'étiquette de la première action à évaluer.

```
/* Déclaration de la
variable globale x */

void incrementer()
{
  x = x +1 ;
  return;
}
void main()
{
  incrementer();
  return;
}
```

(a) Code C

```
main ==
\/
 /\ main_label = "lbl_11"
 /\ IF incrementer_label = "done"
    THEN
     /\ main_label' = "lbl_12"
     /\ incrementer_label' = "lbl_6"
     /\ UNCHANGED <<x>>
    ELSE
     /\ incrementer
     /\ UNCHANGED <<main_label>>
\/
 /\ main_label = "lbl_12"
 /\ main_label' = "done"
 /\ UNCHANGED <<x>>
```

(b) Code TLA+

FIGURE 3: Traduction d'un appel de fonction

La FIGURE 3 présente un bout de code d'une fonction `main` qui appelle la fonction sans arguments `incrementer`. Dans cette figure, nous donnons seulement la traduction de l'appel de fonction dans `main`. Il est à noter que l'appel de fonction peut se faire à plusieurs endroits étant donné que la fonction appelée est traduite indépendamment de l'appelant.

3) Retour de fonction: L'exécution d'une fonction C se termine soit lorsque l'accolade fermante est atteinte, soit lorsque le mot-clé `return` est rencontré. La valeur renvoyée par une fonction est donnée comme paramètre à `return`. Quand la fonction ne renvoie rien, `return` est appelé sans paramètre. Le mot-clé `return` correspond dans TLA+ à la dernière action à évaluer. Quand la fonction renvoie un paramètre, nous définissons une nouvelle variable dont l'identificateur est de la forme `nom_fonction_return` et qui prend la valeur de retour de la fonction. La FIGURE 3 présente la traduction d'un retour de fonction sans paramètre.

La traduction se fait en définissant une conjonction de formules :

– Une formule définissant l'étiquette de l'action actuelle,
– Une formule définissant l'étiquette de l'action à évaluer dans le prochain état qui prend la valeur `"done"`,
– Une formule définissant les variables dont les valeurs sont inchangées au prochain état. Si le retour de fonction se fait avec valeur, nous définissons une formule qui affecte la valeur de retour de la fonction à la variable `nom_fonction_return`.

D. Construction de la spécification

Une spécification TLA+ doit s'écrire sous la forme :

$$Spec \triangleq Init \wedge \Box[Next]_{vars} \wedge Temporal$$

Toutes les variables dans TLA+ doivent être initialisées. Il est donc nécessaire de définir le prédicat `Init` qui initialise toutes les variables déclarées dans le module TLA+. La FIGURE 4 présente la traduction d'un bout de code C formé de déclarations de variables globales et locales avec et sans initialisations. `Init` est formé d'une conjonction de formules dont chacune est une affectation de valeur à chaque variable. Pour les variables non initialisées, nous traduisons leurs initialisations en leur affectant un domaine de valeurs dépendant de leur type. Les bornes du domaine sont modifiables directement en fonction des propriétés à vérifier sur le système. Dans l'exemple de la FIGURE 4, nous affectons à la variable `x` et `main_u` le domaine `(0..10)`. La fonction `main` constitue

```
int x = 0;
int y;
void main() {
  int u;
  /*Liste d'instructions*/ }
```

(a) Code C

```
Init ==
  /\ x = 0
  /\ y \in (0..10)
  /\ main_u \in (0..10)
  /\ main_label = "lbl_5"
```

(b) Code TLA+

FIGURE 4: Traduction de l'initialisation des variables

le programme principal en C. Toutes les instructions et les appels de fonctions sont contenus dans celle-ci. Pour simuler cet aspect dans TLA+, nous spécifions le prédicat `Next` (voir FIGURE 5), qui définit les transitions possibles du programme comme étant la disjonction du prédicat qui appelle la fonction `main` et la formule qui énonce que l'état dernier de la spécification est celui pour laquelle le système se trouve dans la dernière action, étiquetée par `"done"` et pour lequel les valeurs des variables restent inchangées.

La spécification du module TLA+ est définie par le prédicat `Spec`, qui s'écrit de la forme suivante :

```
Spec == Init /\[][Next]_vars
```

```
Next ==
  \/ main
  \/ (main_label = "done" /\ UNCHANGED vars )
```

FIGURE 5: Prédicat $Next$ du module TLA+

Il est à noter que dans ce papier nous ne traitons pas les propriétés temporelles à vérifier sur le système, qui sont définies par le prédicat `Temporal`.

V. IMPLÉMENTATION

Notre traducteur est implémenté avec la plateforme Frama-C [3] et consiste en plusieurs fichiers OCaml faisant en tout 700 lignes de code. En effet, le noyau de Frama-C interagit avec la bibliothèque CIL [16] pour normaliser le code dans un premier temps. Ensuite, il parse le code pour fournir un arbre de syntaxe abstraite (AST), qui est la représentation interne des caractéristiques essentielles du programme C. Une fois l'AST généré, des visiteurs intégrés dans Frama-C sont utilisés par le traducteur pour visiter les nœuds de l'arbre et générer le code TLA+.

```
int n;
void suiteCollaz()
{
  while (n != 1)
  {
    if (n % 2 == 0)
      n/=2;
    else n = 3*n+1;
  }
  return;
}
void main()
{
  suiteCollaz();
  return;
}
```

FIGURE 6: Programme C de l'algorithme de Collatz

L'étude de cas a porté sur l'algorithme de Collatz [10] dont le code C associé est donné par la FIGURE 6.
Dans cet algorithme, nous partons d'un nombre entier plus grand que zéro ; s'il est pair, on le divise par 2 ; s'il est impair, on le multiplie par 3 et on ajoute 1. En répétant l'opération, nous obtenons une suite d'entiers positifs cyclique, puisque l'obtention de la valeur `1` fait boucler indéfiniment l'algorithme.

```
---------- MODULE collatz ----------
EXTENDS Naturals, TLC
VARIABLES n,suiteCollaz_label,
          main_label
-----------------------------------
vars == <<n,suiteCollaz_label,
          main_label>>
Init ==
    /\ n \in (2..100)
    /\ suiteCollaz_label = "lbl_4"
    /\ main_label = "lbl_14"

suiteCollaz ==
\/
  /\ suiteCollaz_label = "lbl_4"
  /\ IF (n /= 1)
     THEN suiteCollaz_label' = "lbl_6"
     ELSE suiteCollaz_label' = "lbl_9"
  /\ UNCHANGED <<n>>
\/
  /\ suiteCollaz_label = "lbl_6"
  /\ IF (n % 2 = 0)
     THEN suiteCollaz_label' = "lbl_7"
     ELSE suiteCollaz_label' = "lbl_8"
  /\ UNCHANGED <<n>>
\/
  /\ suiteCollaz_label = "lbl_7"
  /\ n' = n \div 2
  /\ suiteCollaz_label' = "lbl_4"
\/
  /\ suiteCollaz_label = "lbl_8"
  /\ n' = 3 * n + 1
  /\ suiteCollaz_label' = "lbl_4"
\/
  /\ suiteCollaz_label = "lbl_9"
  /\ suiteCollaz_label' = "done"
  /\ UNCHANGED <<n>>

vars_suiteCollaz ==
    <<suiteCollaz_label>>

main ==
\/
  /\ main_label = "lbl_14"
  /\ IF suiteCollaz_label = "done"
     THEN
       /\ main_label' = "lbl_15"
       /\ suiteCollaz_label' = "lbl_4"
       /\ UNCHANGED <<n>>
     ELSE
       /\ UNCHANGED <<main_label>>
       /\ suiteCollaz
\/
  /\ main_label = "lbl_15"
  /\ main_label' = "done"
  /\ UNCHANGED <<n,vars_suiteCollaz>>
-----------------------------------
Next ==
  \/ main
  \/ (main_label = "done"
     /\ UNCHANGED vars )

Spec == Init /\ [][Next]_vars
===================================
```

FIGURE 7: Traduction de l'algorithme de Collatz

La traduction de cet algorithme en TLA+ est donnée par la FIGURE 7. Le code TLA+ permet de générer toutes les suites possibles pour les valeurs de `n` entre `2` et `100`. Parmi les propriétés à vérifier sur ce code est la propriété de terminaison. Cette propriété s'écrit comme suit : `[](Spec => (main_label = "done" /\ n = 1))`, autrement dit que l'on finit toujours par trouver la valeur `1` au fil des calculs quel que soit l'entier de départ. Cette propriété est vérifiée par TLC. D'autres propriétés sont énoncées dans [10] mais qui ne font pas l'objet de notre étude dans ce papier.

VI. CONCLUSION

Dans ce court article, nous avons présenté un processus de transformation d'un code source C vers le langage TLA+. La proposition développée consiste à rendre la phase de modélisation automatique pour éviter les fautes pouvant survenir au cas où la modélisation se ferait manuellement d'une part et faciliter la tâche au concepteur durant la phase de vérification du système d'autre part. Ce travail constitue une première version du traducteur, nous envisageons d'abord de le compléter pour prendre en compte les autres aspects du langage C, notamment les pointeurs et les fonctions récursives. Ensuite, l'intégrer dans une suite d'outil pour l'aide à la conception et la vérification des systèmes.

RÉFÉRENCES

[1] Modex home page. http ://cm.bell-labs.com/cm/cs/what/modex/.

[2] Iso/iec 9899 :1990 : Programming languages-c, 1990.

[3] Framework form modular analysis of c. http ://www.frama-c.cea.fr, 2008.

[4] Thomas Ball and Sriram K. Rajamani. The SLAM project : debugging system software via static analysis. *SIGPLAN Not*, 2002.

[5] Edmund M. Clarke and Jeannette M. Wing. Formal methods : state of the art and future directions. *ACM Comput. Surv.*, 28(4) :626–643, 1996.

[6] Klaus Havelund and Thomas Pressburger. Model Checking JAVA Programs using JAVA PathFinder. *STTT*, 2(4) :366–381, 2000.

[7] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Gregoire Sutre. Software Verification with BLAST. pages 235–239. Springer, 2003.

[8] Gerard J. Holzmann. The Model Checker SPIN. *IEEE Trans. Software Eng.*, 23(5) :279–295, 1997.

[9] Gerard J. Holzmann. Logic verification of ANSI-C Code with SPIN. In *SPIN*, pages 131–147, 2000.

[10] Neil D. Jones. Abstract interpretation : a semantics-based tool for program analysis, 1994.

[11] Bengt Jonsson Ke Jiang. Using SPIN to model check concurrent algorithms, using a translation from C to Promela. In *Proc. 2nd Swedish Workshop on Multi-Core Computing*, 2009.

[12] Leslie Lamport. How to write a long formula (short communication). *Formal Asp. Comput.*, 6(5) :580–584, 1994.

[13] Leslie Lamport. The Temporal Logic of Actions. *ACM Trans. Program. Lang. Syst.*, 16(3) :872–923, May 1994.

[14] Leslie Lamport. *Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002.

[15] Markus Müller-Olm, David A. Schmidt, and Bernhard Steffen. Model-Checking : A Tutorial Introduction. In *Static Analysis, 6th International Symposium, SAS 99, Venice, Italy*.

[16] George C. Necula, Scott Mcpeak, Shree P. Rahul, and Westley Weimer. CIL : Intermediate language and tools for analysis and transformation of C programs. In *International Conference on Compiler Construction*, pages 213–228, 2002.

[17] Yuan Yu, Panagiotis Manolios, and Leslie Lamport. Model checking TLA+ specifications. In *Correct Hardware Design and Verification Methods*, pages 54–66. Springer-Verlag, 1999.