Translating C to PlusCal for Model Checking of Safety Properties on Source Code

GUILLAUME DI FATTA, AMIRA METHNI,
EMMANUEL OHAYON

Asterios Technologies
Core Team

May 2025

SAFRAN
CentraleSupélec

Context

- Student in final year of MSc at CentraleSupelec/Paris-Saclay University
- Gap year, two internships of 5 months to do.
- **Internship Subject** : Modelization and formal verification, with TLA+ and TLC, of real-time synchronous algorithms of Asterios Technologies Micro-Kernel.
- Supervised by Emmanuel OHAYON and Amira METHNI.

Summary

1. Introduction and Context.
2. Tool Presentation : C2PlusCal.
3. Implementation Details.
4. How to manage memory.
5. Results, limitations and perspectives.
6. Conclusion

Part 1

Introduction and Context

Context

- **Asterios Technologies** : Subsidiary of Safran Electronics and Defense
- Provides software solutions to orchestrate, integrate and certify critical real-time applications
- These solutions run on embedded systems with a Micro-Kernel developed by Asterios Technologies
- Idea to use TLA+ to verify multi-core algorithms, which are real-time and critical

Context

- Use of TLA+ for multiple purposes and experimentation, such as classical formal specification but also to write specification from the source code.
- Verification of the Scheduler of the kernel :
    - Verify simultaneously various configurations at once
    - Avoid manual test, which could not be exhaustive, and long
- The idea is to see what can be achieved and experiment, not to use it directly in the company verification process

→ Translation directly from the code.

This approach is possible, because the C code is simple and embedded on a micro-kernel (no libc, no dynamic memory, etc)

Motivations

Main motivations :

- Previous work : *« C2TLA+ : Automated translation from C code to TLA+ »* (Amira Methni).
- Relatively concise code, with a simple and clear structure.
- First successful small translations by hand

We wanted to revive and rehabilitate this old project, with the main difference to translate to PlusCal instead of TLA+.

Context

Objectives :

- Translation to PlusCal, to use the language paradigms closer to C than TLA+
- Transpilation of the whole Scheduler source code
- Find a known bug in an old version, with TLC Model-Checking
- Describe invariants on the implementation to verify code properties

Automatize the translation of C programs in PlusCal, to facilitate their verification.

Part 2

Tool Presentation : C2PlusCal

Frama-C

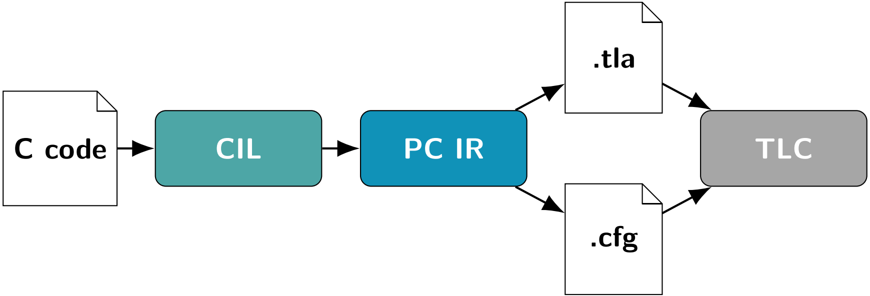
- **Frama-C** : set of interoperable program analyzers for C programs
- Used to pre-processed C program to CIL representation and retrieve the AST

Frama-C

- **Frama-C** : set of interoperable program analyzers for C programs
- Used to pre-processed C program to CIL representation and retrieve the AST
- **OCaml** : Functional language, used to make a Frama-C plug-in
- New personalized IR to translate to PlusCal

Frama-C

- **Frama-C** : set of interoperable program analyzers for C programs
- Used to pre-processed C program to CIL representation and retrieve the AST
- **OCaml** : Functional language, used to make a Frama-C plug-in
- New personalized IR to translate to PlusCal



Program Representation

- C Program is a collection of global functions and variables
- We will use sequences to represent memory

Program Representation

- C Program is a collection of global functions and variables
- We will use sequences to represent memory
- Functions are translated into procedures
    - Convenient for arguments and automatic return flow
    - Does not support return values

Program Representation

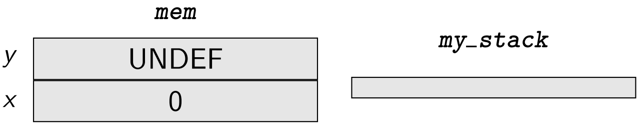
- C Program is a collection of global functions and variables
- We will use sequences to represent memory
- Functions are translated into procedures
  - Convenient for arguments and automatic return flow
  - Does not support return values
- Different memory locations
  - A memory shared between procedures : mem
  - A memory used for locals : my_stack
  - A memory used for return values : ret

We write a naive C interpreter with a stack in PlusCal

Program Example

Code C

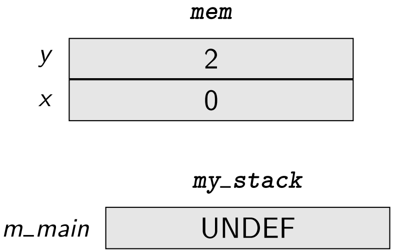
```c
int x = 0;
int y;
int mean(int a, int b)
{
    return a + b / 2;
}
int main()
{
    y = 2;
    int m = mean(x, y);
    return 0;
}
```



Program Example

Code C

```c
int x = 0;
int y;
int mean(int a, int b)
{
    return a + b / 2;
}
int main()
{
    y = 2;
    int m = mean(x, y);    ←
    return 0;
}
```



mem

y 2

x 0

my_stack

m_main UNDEF

Program Example

Code C

```c
int x = 0;
int y;
int mean(int a, int b)
{
    return a + b / 2;
}
int main()
{
    y = 2;
    int m = mean(x, y);
    return 0;
}
```

mem

y	2
x	0

my_stack

b_mean	2
a_mean	0
m_main	UNDEF

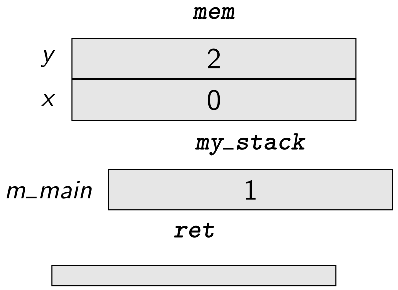
ret

1

Program Example

Code C

```c
int x = 0;
int y;
int mean(int a, int b)
{
    return a + b / 2;
}
int main()
{
    y = 2;
    int m = mean(x, y);
    return 0;
}
```



mem

y 2

x 0

my_stack

m_main 1

ret

Program Representation

- Variables will be represented as pointers
- PlusCal macros are defined to load/store from the memory

Program Representation

- Variables will be represented as pointers
- PlusCal macros are defined to load/store from the memory

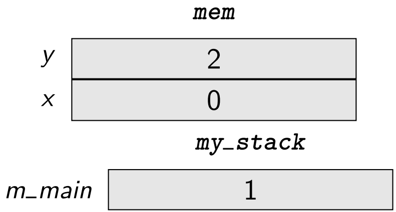
A variable is a record with three fields :

- loc : The memory region where it is stored
- fp : Frame pointer in this region
- offs : Offset from frame pointer

Program Example

Code C

```c
int x = 0;
int y;
int mean(int a, int b)
{
    return a + b / 2;
}
int main()
{
    y = 2;
    int m = mean(x, y);
    return 0;
}
```



mem

y | 2

x | 0

my_stack

m_main | 1

$$y\_glob\_ptr = [loc\ |\text{->}\ \text{"mem"},\ fp\ |\text{->}\ 0,\ offs\ |\text{->}\ 1]$$

$$m\_main\_ptr = [loc\ |\text{->}\ \text{"my\_stack"},\ fp\ |\text{->}\ 0,\ offs\ |\text{->}\ 0]$$

Program Representation

- Variables will be represented as pointers
- PlusCal macros are defined to load/store from the memory

Several macros are defined :

- load : to retrieve a value from a stack, defined as TLA+ operator because it needs to represent a value
- store : to put a value in a stack
- ret_attr : to retrieve the last returned value, used for ”a = f(x)”
- decl : to initialize the pointer at the right value and add it on the stack

Program Translation Example

C Code

```c
// Function to add 2 to a given number
int add_two(int x) {
    int y = 2;
    x += y;
    return x;
}
```

PlusCal

```
procedure add_two(x_add_two)
variables
    x_ptr_add_two = [loc |-> "stack", fp |-> Len(my_stack), offs
          |-> 0];
    y_ptr_add_two = [loc |-> "stack", fp |-> Len(my_stack), offs
          |-> 0];
begin
    Line0_add_two:
    decl(x_add_two,x_ptr_add_two);
    decl(UNDEF,y_ptr_add_two);

    Line2_add_two:
    store(2,y_ptr_add_two);
    store((load(my_stack, x_ptr_add_two)+load(my_stack,
          y_ptr_add_two)),x_ptr_add_two);
    push(ret, load(my_stack, x_ptr_add_two));

    Line5_add_two:
    pop(my_stack);
    pop(my_stack);
    return;
end procedure;
```

Program Representation

- C global variables are initialized in a separated process
- A PlusCal global flag is used to prevent the beginning of other processes before the end of initialization
- For the moment, only one other process, which calls the entry function of C program

Program Representation

```
fair process globalInit \in GLOBAL_INIT
variables
begin
    \* Initialization of global variables
    initDone := TRUE;

end process;
fair process proc \in PROCESS
variables
begin
    Line0_proc:
    await initDone = TRUE;
    Line1_proc:
    call main();
end process;
```

Part 3

Implementation Details

CIL Representation

- Pre-processes C program
- Transforms any loop in *while(1)* with **break** labels and **goto**
- Expressions that contain side-effects are separated into statements

CIL Representation

This helps us to make direct translation from C to PlusCal :

Code PlusCal

```
Line54_main:
while(TRUE) do
    Line54_main0:
    if((load(my_stack, j_ptr_main)<10)) then
        Line54_main00:
        skip;
    else
        Line54_main01:
        goto while_1_break;
    end if;
    Line54_main1:
    store((load(my_stack, d_ptr_main)+load(my_stack,
          i_ptr_main)),d_ptr_main);
    Line54_main2:
    store((load(my_stack, j_ptr_main)+1),j_ptr_main);
end while;
while_1_break:
skip;
```

Code C

```
for (int j = 0; j < 10; j++) {
    d += i;
}
```

Complex Data Structures

We translate directly complex data structures :

- Arrays are represented as sequences
- Structures are represented as records

Complex Data Structures

We translate directly complex data structures :

- Arrays are represented as sequences
- Structures are represented as records

Code C

```
    struct Error {
    char* name;
    int id;
     };

     struct Error global_error = {"test global
          error", 1};
```

Code PlusCal

```
    global_error := [name |-> "test global error",
         id |-> 1];
```

Complex Data Structures

- Static context, thus arrays are already allocated with a fixed size
- Uninitialized arrays are filled with *UNDEF*

Complex Data Structures

- Static context, thus arrays are already allocated with a fixed size
- Uninitialized arrays are filled with *UNDEF*

Code PlusCal

```
procedure init_array(size, arr_ptr) begin
  InitArray:
  tmpArrayFill := 0;
  store(<<>>, arr_ptr);

  WhileInitArray:
  while(tmpArrayFill < size) do
    store(Append(load(my_stack, arr_ptr), UNDEF), arr_ptr);
    tmpArrayFill := tmpArrayFill + 1;
  end while;

  return;
end procedure;
```

Pointer Arithmetic

- Our pointer representation allows pointer arithmetic
- All data types have size 1
- We can add int to pointers
- We can add two pointers

Pointer Arithmetic

- Our pointer representation allows pointer arithmetic
- All data types have size 1
- We can add int to pointers
- We can add two pointers

Code C

```
    int x = 1;
    int* x_ptr = &x;
    x_ptr += 1;
```

Code PlusCal

```
    store(1,x);
    store(x,x_ptr);
    store([x_ptr EXCEPT !.offs = @+1], x_ptr);;
```

Part 4

How to manage memory

Current issues

The representation of complex types as sequences/records :

- Simplifies translations
- Allows all data types to have size 1

Current issues

The representation of complex types as sequences/records :

- Simplifies translations
- Allows all data types to have size 1

The current pointer representation for variables has several limitations :

- It prevents taking the address of struct fields or array elements
- It does not always correctly handle expressions like `**x`

These issues become particularly problematic when dealing with nested structures or arrays involving pointers

New approach

An other representation for pointer representation can be used :

- New fields
  - ptr : pointer that points where the element is stored
  - ref : a reference to the target element, such as a field name or an array index

New approach

An other representation for pointer representation can be used :

- New fields
    - ptr : pointer that points where the element is stored
    - ref : a reference to the target element, such as a field name or an array index

Code C

```c
int** ptr_field = &(error_ptr->id);
```

Code PlusCal

```
store([ptr |-> load(my_stack,error_ptr), ref |->
    "id"], ptr_field);
```

Advantages

- Allows to take addresses from new elements
- Simplifies and homogenizes access to arrays elements

Advantages

- Allows to take addresses from new elements
- Simplifies and homogenizes access to arrays elements

Code C

```
    array[x] = 3;
```

Code PlusCal

```
    load(my_stack, array_ptr)[x] := 3;
```

Advantages

- Allows to take addresses from new elements
- Simplifies and homogenizes access to arrays elements

Code C

```
    array[x] = 3;
```

Code PlusCal

```
    store(3, [ptr |-> array_ptr, ref |-> x]);
```

New approach

Macros are defined to support both pointer representations :

```
    RECURSIVE load(_,_)
    load(stk, base) == IF "ptr" \in DOMAIN base THEN
                         load(stk, base.ptr)[base.ref]
                      ELSE
                         IF base.loc = "stack"
                         THEN stk[Len(stk) - (base.fp + base.offs)]
                         ELSE mem[Len(mem) - base.offs]
```

New approach

Macros are defined to support both pointer representations :

```
RECURSIVE load(_,_)
load(stk, base) == IF "ptr" \in DOMAIN base THEN
                     load(stk, base.ptr)[base.ref]
                  ELSE
                     IF base.loc = "stack"
                     THEN stk[Len(stk) - (base.fp + base.offs)]
                     ELSE mem[Len(mem) - base.offs]
```

- Recursive definition to trace back to the base pointer of an expression

New approach

Store macro is redefined as well :

```
RECURSIVE idx_seq(_,_)
idx_seq(stk, base) == IF "ptr" \in DOMAIN base THEN
                         idx_seq(stk, base.ptr) \o <<base.ref>>
                      ELSE
                         IF base.loc = "stack"
                         THEN <<"stack", Len(stk) - (base.fp + base.offs)>>
                         ELSE <<"mem", Len(mem) - base.offs>>
RECURSIVE update_stack(_,_,_)
update_stack(stk, val, seq) == IF seq = <<>>
                                THEN val
                                ELSE [stk EXCEPT ![seq[1]] = update_stack(stk[seq[1]], val, Tail(seq))]
```

```
macro store(val, ptr) begin
  with seq = idx_seq(my_stack, ptr) do
  if seq[1] = "stack"
    then my_stack := update_stack(my_stack, val, Tail(seq));
    else mem := update_stack(mem, val, Tail(seq));
  end if;
  end with;
end macro;
```

New approach

```
macro store(val, ptr) begin
  with seq = idx_seq(my_stack, ptr) do
  if seq[1] = "stack"
    then my_stack := update_stack(my_stack, val, Tail(seq));
    else mem := update_stack(mem, val, Tail(seq));
  end if;
  end with;
end macro;
```

- Use of *idx_seq* operator to retrieve the sequence of indexes to access the element
- Use of *update_stack* operator to update the corresponding stack

Results, limitations and perspectives

Experimental Results

Experimental results on the scheduler source code :

- The bug was found with a handwritten translation
- $\approx 45 \mathit{min}$ of verification with TLC on 10 cores of an Intel Core Ultra 9 185H
- The proposed invariant is intuitive and could have been written without prior knowledge of the bug

Experimental Results

Experimental results on the scheduler source code :

- The bug was found with a handwritten translation
- $\approx 45 min$ of verification with TLC on 10 cores of an Intel Core Ultra 9 185H
- The proposed invariant is intuitive and could have been written without prior knowledge of the bug

There are still several limitations :

- Difficulty of reading and properties writing
- We wrote arbitrary abstractions, that lack proximity to the original source code

Limitations and perspectives

Current limitations :

- Explosion of the state space, because each line has a PlusCal label
- **Labels** used by Frama-C may have duplicated names in the presence of multiple loops or nested if/else statements
- Incomplete management of certain pointer operations
- Syntactical construction not handled :
  - **Keywords** : `typeof`, `sizeof`, `switch`, etc.

Limitations and perspectives

Current limitations :

- Explosion of the state space, because each line has a PlusCal label
- **Labels** used by Frama-C may have duplicated names in the presence of multiple loops or nested if/else statements
- Incomplete management of certain pointer operations
- Syntactical construction not handled :
  - **Keywords** : `typeof`, `sizeof`, `switch`, etc.

Possible Evolutions :

- Handle different threads in different PlusCal processes
- Open-source project <3

To Conclude

Conclusion :

- The initial goal was to play with PlusCal/TLA+ and see what can be done
- Automated translation is still an experimental approach but shows promising results

To Conclude

Conclusion :

- The initial goal was to play with PlusCal/TLA+ and see what can be done
- Automated translation is still an experimental approach but shows promising results

Perspectives :

- Exploration of **TLAPS**
- Use of Apalache instead of TLC to compare performance
- Show that the translated specification refines an abstract specification

To Conclude

- A big thank you to **Asterios Technologies** for their warm welcome and support throughout this internship
- Special thanks to **Emmanuel Ohayon** and **Amira Methni** for their guidance, availability, and advice
- If you'd like to stay in touch or check out some of my projects : LinkedIn (Guillaume DI FATTA) / GitHub (Atafid)

Thank you !

Thanks for your attention ! :)

Do you have any question ?

Références

- Leslie Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Available on : `https://lamport.azurewebsites.net/tla/book-21-07-04.pdf` Microsoft Research, 2002.
- Amira Methni, Matthieu Lemerre, Belgacem Ben Hedia, Kamel Barkaoui, Serge Haddad. *Specifying and Verifying Concurrent C Programs with TLA+. In: Artho, C., Ölveczky, P. (eds) Formal Techniques for Safety-Critical Systems. FTSCS 2014. Communications in Computer and Information Science, vol 476. Springer, Cham*. Available on : `https://doi.org/10.1007/978-3-319-17581-2_14`
- Guilaume Di Fatta, Amira Methni, Emmanuel Ohayon. *Github of C2PlusCal*. Available on : `https://github.com/asterios-technologies/c2pluscal`