Available online to ejournal.pnc.ac.id

Journal of Innovation Information Technology and Application

Journal Page is available to https://ejournal.pnc.ac.id/index.php/jinita

Towards Automatic Maude Specifications Generation From C Functions

Fateh Boutekkouk [1]

[1]*Department of Computer Science, University of Larbi Ben M'hedi, Oum El Bouaghi, Algeria,*

email:[1] *Fateh_boutekkouk,ibnm@yahoo.fr*

ARTICLE INFO

Article history:
Received 05 April 2023
Revised 26 June 2023
Accepted 26 June 2023
Available online 26 June 2023

Keywords:
C programming language
Maude
Formal specification
Formal verification

IEEE style in citing this article:
F. Boutekkouk, "Towards Automatic Maude Specifications Generation From C Functions," Journal of Innovation Information Technology and Application (JINITA), vol. 5, no. 1, pp. 83–96.

ABSTRACT

In this paper, we aim to contribute to the knowledge about how imperative C functions can be transformed to Maude functional and system modules respectively. Maude is a formal specification language characterized by simplicity, expressivity and good performance. It is a multi-paradigm meta-language based on rewriting logic and equational theories used to specify, simulate and formally verify concurrent and distributed systems. Maude has been used to define the operational semantics of many programming and specification languages. In particular, the addition of this paper is to close the gap between a subset of the C standard language and Maude relying on a transformational approach.

1. INTRODUCTION

Since its earlier occurrence, the C language remains among the most widely used programming languages in software coding. This is due to the fact that C provides programmers with a great deal of flexibility in the representation of data and the use of pointers.

However, C is also among the most bug prone programming languages because C is not strongly-typed. The official specification of the C language (The ISO C standard [1]) underspecifies the semantics of the C language in three different ways: Implementation-defined, unspecified, undefined. As an example of implementation-defined behavior is the propagation of the high-order bit when a signed integer is shifted right. As an example of unspecified behavior is the order, in which the arguments to a function and arithmetic expressions are evaluated. These behaviors lead to side effects because it depends on the compiler used. As an example of undefined behavior is dereferencing a NULL pointer or integer overflow. This behavior leads to software run time bugs and security issues.

On the other hand, the functional programming paradigm [2] is a well known programming model that has been used in the context of formal computing and Artificial Intelligence.

The functional and imperative paradigms have different viewpoints. The functional paradigm is based on the function mathematical concept. A function evaluates an expression to produce a value and the whole program can be seen as a composite function. The order in which subexpressions are evaluated does not affect the resulting values. Consequently, the functional paradigm is most suitable for deterministic systems without side effects. The imperative paradigm however, relies on the execution of statements that change the state of the memory. The order in which statements are executed affects the result. The imperative paradigm is most suitable for nondeterministic systems with side effects [3].

Our idea is to transform C functions without side effects to Maude functional modules defining equational theory and C procedures (i.e. functions without return value) with side effects to Maude system

**) Corresponding Author : Fateh_boutekkouk,ibnm@yahoo.fr*

modules defining rewriting logic theory. In fact, the literature on formal verification of C code is very rich [1, 2, 5, 6, 11, 12, 13, 15, 16, 17, 18, 19]. However, in this work we are interested in the transformation of a C code to a Maude code.

Maude [4] is a formal specification language characterized by simplicity, expressivity and good performance. It is a multi-paradigm meta-language based on rewriting logic and equational theories used to specify, simulate and formally verify concurrent and distributed systems. Maude has been used to define the operational semantics of many programming and specification languages [3, 20].

The move towards functional paradigm enables formal proofs [7]. Similarly, narrowing C procedures to rewriting rules will permit to leverage the rewriting logic capabilities for reachability analysis, symbolic execution and model checking. Thanks to Maude increasing tools, one can mathematically reason about functional modules with respect to some interesting properties as termination, Ross church property and completeness and system modules with regard to reachability analysis and other user-defined properties using Maude LTL model checker. In this work, we will be interested in C functions and procedures. C functions can include any type of statement without side effects, conditional statements as if and if-else statements, and while loops. Instead, C procedures can include any form of side-effect statements.

Maude, on the other side, is qualified as a high-level functional programming/specification language. It does not rely on variables assignments, rather than, it relies on equational algebra. Equations are axioms defining the way calculations are performed. They do not use neither local/global variables nor loops. Instead, they use recursive and higher order functions.

Another issue with the transformation of a C procedure (i.e. a void function) to rewriting logic theory is that a C procedure is sequential; however, the rewriting logic is by principle concurrent. Through this research work, we investigate the possibility of transformation of a subset of the C language to a subset of the Maude language. As a first attempt, we tried to make the transformation as simple as possible while closing to maximum the semantic gap between the two languages by choosing carefully the Maude concepts that match well the semantic of a restricted set of C constructors. This paper is structured as follow: in section two the fundamental concepts of the Maude language are presented. In section three, we detail the transformation of C functions with returned value to Maude functional modules with some illustrative examples. In section three, we discuss the transformation of C functions without parameters to Maude functional modules. Section four is dedicated to the transformation of C procedures to Maude system modules. In section five, some reduction and rewriting results are presented. We close this paper by a conclusion and some short-term perspectives.

2. Maude language

Maude is a high-level formal specification and programming language supporting equational, rewriting logic and object oriented computations offering a unified logical framework for verification and validation of concurrent and distributed systems. Simplicity, expressivity and good performance are the three key characteristics of the Maude system. In addition, Maude is reflexive in the sense that its concepts are specified at Maude meta-level using Maude itself. Maude supports the meta-programming and parametrized programming as well. Rewriting logic is the logic of concurrent change. It is a flexible and general semantic framework for giving semantics to a wide range of languages and models of concurrency.

The equational logic underlying Maude is membership equational logic. In this logic sorts (i.e. types) are grouped into equivalence classes called kinds. Maude sorts are user-defined, while kinds are implicitly associated with connected components of sorts and are considered as error supersorts. The specification unity in Maude is called module. In Maude, there exist three types of modules: functional modules defining equational theory, system modules defining rewriting logic theory and object modules implementing some of the object oriented programming paradigm principles as classes, objects, messages, heritage and polymorphism. The same Maude module can simultaneously be viewed as an executable formal specification and as a program. nonexecutable specifications can also be defined in a Maude. Such specifications are generally theorems and lemmas used for the purpose of theorem proving.

A functional module has the syntax: fmod ModuleName is [sorts, operations signature, membership axioms, equations] endfm.

A functional module defines sorts, operations on these sorts and their equations and/or membership axioms. Listing (a) shows an example of a functional module named PEANO-NAT-EXTRA. It defines a sort Nat, two constructors 0 and s and an operation + with its equation.

DOI: doi.org/10.35970/jinita.v5i1.1846

```
fmod PEANO-NAT-EXTRA is
sorts Nat NzNat .
subsort NzNat < Nat .
op 0 : -> Nat [ctor] .
op s : Nat -> Nat [ctor iter] .
op _+_ : Nat Nat -> Nat .
vars M N : Nat .
eq 0 + N = N .
eq s(M) + N = s(M + N) .
endfm
```

(a)

Sorts are declared using the keyword sort or sorts. A sort can be interpreted as a set of values. The keyword subsort or subsorts define a sub-set of sorts. To each declared sort, Maude creates automatically its corresponding kind. To make difference between a sort and a kind, the syntax [sort] is used to denote a kind. All subsorts of a sort form one component having the same kind. Kinds can be used to define partial operations. An operation signature is declared using the keyword op or ops and may have optional attributes. Each operation signature includes the definition domain (arguments sorts), and co-domain or rang (result sort). An operation without arguments is a constant. The attribute ctor is used to define a constructor. An operation can be notated using prefix or mixfix notation. The latter requires to specify explicitly the position of arguments using the underscore character. A Maude operation can be overloaded but the most famous overloading is subsort overloading. Variables (i.e. operations parameters) are declared using the keyword var or vars. An equation is defined using the keyword eq and may have a set of optional attributes. An equation has the form eq t = t' where ta and t' are two terms of the same kind. Note that variables are not allowed to figure in t' if they do not figure in t. A conditional equation is defined using the keyword ceq. A condition can be an equation, a matching equation having the form t := t' or a Boolean equation. A Maude module can import other modules using one of the three available importation modes: protecting, extending or including. Equations are reduced (i.e. evaluated) using the command 'reduce' or 'red'. Maude includes many predefined functional modules (i.e. in-built modules) as BOOLEAN, INT, FLOAT, STRING, RATIONAL, LIST, CONVERSION and others. Furthermore, Maude offers a variety of tools for verification of inductive properties using the Maude's inductive theorem prover (ITP), verification of termination using the Maude termination tool (MTT), verification of the church-Rosser property using the Maude church-Rosser checker (CRC) and verification of completeness using the Maude Sufficient Completeness Checker (SCC).

A system module has the syntax: mod ModuleName is [sorts, operations, equations, rewriting rules] endm

```
mod CIGARETTES is
sort State .
op c : -> State [ctor] . *** cigarette
op b : -> State [ctor] . *** butt
op __ : State State -> State [ctor assoc comm] .
rl [smoke] : c => b .
rl [makenew] : b b b b => c .
endm
```

(b)

In fact, a system module is an extension of functional module to support rewriting logic and rewriting logic rules are rewritten modulo equations. listing (b) shows a Maude system module named CIGARETTES. It defines two rewriting rules labeled [smoke] and [makenew]. In its general form, a rewriting rule has the form t -> t' where t and t' are two terms of the same kind. The two rewriting rules are rewritten modulo the operation of concatenation op __ : State State -> State [ctor assoc comm] . assoc and comm denote associative and commutative. All rewriting rules are in essence concurrent and each rule models a transition with a partial state. Rewriting rules can be conditional. The condition can be an equation, a rewriting rule, or a membership condition having the form t :: sort. As in equations, rewriting rules may have optional attributes. Rewriting rules are reduced using the command rewrite or rew. Other commands exist. The command frewrite is used for Fairless rewriting. The difference between the two commands resides in the rewriting strategy. A user-defined strategy can also be defined at the Meta level. The command rew may have some optional parameters to specify for example the maximum number of rewriting steps. Beside the command rew, Maude offers the command search for reachability analysis and

an LTL model checker to check user-defined properties expressed in Linear Temporal Logic LTL. Maude also includes a SAT solver, unification and narrowing tools. Unification is a technique used to solve equations based on substitutions. Narrowing is an extension of rewriting theory where free variables are allowed to figure in terms and unified via matching and substitutions used for symbo lic reachability analysis. In spite of all these big advantages, Maude still lack a link to some common programming languages as C and Java. Our ultimate objective is to develop a tool permitting automatic Maude code generation from C code and vice versa. This paper traces the big lines for automatic Maude code generation from C code. To our best knowledge, this work is the first one trying to establish a bridge between Maude and C languages following a transformational approach.

3. Transformation of C functions with returned values to Maude functional modules

3.1. Function signature

In C, the signature of a function returning a value has the form: type function f(type_1 p1, type_2 p2, …, type_n pn) where type is the type of the returned value and p1, …pn, are the function parameters having types type_1, type_2, …, type_n respectively. In Maude: op f : type_1 type_2… type_n -> type . where type, type1, type2, …typen, can be any Maude in-built sorts as Nat, Int, Float or user-defined sorts declared using the sort keyword. Another possible transformation is : op f : [type_1] [type_2]… [type_n] -> [type] . where [type_1], [type_2],… [type_n] , [type] are kinds and can be used to define partial operations. Other forms of the C function signature are:

type* function f(type_1 p1, type_2 p2, …, type_n) and type** function f(type_1 p1, type_2 p2, …, type_n), Here, *type (resp. **type) means that function f returns a pointer to a 1D array (resp. 2D array) or to some other user-defined types as structures. In these cases, *type will be defined as a Maude list or any user-defined sort.

3.2. Function body

A C function body is composed of a sequence of assignments and control statements and return one value. Input parameters are by default passed by value. A C function may include side effect assignment as x = (x =0) + (x = 1) or ++x + ++x. Another source of side effects in C is the access to global variables and aliased pointers. Such side effect assignments can lead to different behaviors given the same input parameters. Consequently, the result of the computation depends strongly on the C compiler used. Side effect should not be appear in pure functional code. We expect from pure functions to be fully deterministic. In order to transform a C function with a return value to a Maude functional module, we need to make some assumptions:

1.We assume that all side effect assignments are removed from the C code and substituted by simple assignments of the form V = E.
2.The C code is free from pointers except for functions returning arrays.
3.All forms of C loops are substituted by the basic while loops.
4.Switch-case statements are removed and replaced by the basic conditional if-else statements.

In order to explain, our ideas, let us start by some C code. For example listing (a) shows a C function with two integer parameters a and b and returns an arithmetic expression. Note that this functional style of C code does not use any local variable.

```
int f(int a, int b)
{return (a*b) + 20;}
          (a)
```

```
int f(int a, int b)
{ int x = a * b ;int y = x + 20 ;
return y;}
          (a)
```

Code (a') represents the pure imperative style of its equivalent functional style in code (a). Of course we can easily transform (a') to (a) by successive substitutions forwardly. However, this technique does not work well in all cases. Take the example of the C code (b) that swaps the values of two variables a and b without using an intermediate variable:

```
int a; int b;
a = a + b;(1)
b = a - b;(2)
a = a - b;(3)
        (b)
```

If we make sequential forward substitutions, we obtain the code:
b = (a+b)-b (by substituting (1) in (2)) leads to b=a (2')
a = (a+b)-a (by substituting (1) and (2') in (3)) leads to a = b

DOI: doi.org/10.35970/jinita.v5i1.1846

The result is incorrect because b is swapped however, a does not. So, each non side effect assignment is transformed to an equation and the entire C function is defined as an equation combining higher order equations. The most inner equation corresponds to the first assignment and the most outer equation corresponds to the last assignment. A C assignment like x = 10 can be transformed to a Maude constant operation of the form:
op x : -> Int [ctor] .
eq x = 10 .
Code (M1) is the Maude code which corresponds to C code (C1) .

```
fmod Cfunction is
protecting INT .
op f : Int Int -> Int .
vars a b : Int .
eq f(a,b) = (a * b) + 20 .
endfm
```
(C1)

```
fmod Cfunction is
protecting INT .
op f : Int Int -> Int .
vars a b : Int .
op fx : Int Int -> Int .
op fy : Int -> Int .
eq fx(a,b) = a * b .
eq fy(a) = a + 20 .
eq f(a,b) = fy(fx(a,b)) .
endfm
red f(5, 6) . ***reduce
```
(M1)

3.2.1. if statement

Each C if statement is transformed to a conditional equation. In code (C2), the statement x=2*y is executed whether the condition (a > 1000) is satisfied or not. However the statement y=a–y is executed only when the condition is verified.

```
int f(int a)
{int y =200;
if a > 1000  y = a - y ;
int x = 2 * y ;
return x;}
```
(C2)

```
fmod Cfunction is
protecting INT .
op f : Int  -> Int .
var a : Int .
op y : -> Int [ctor] .
eq y = 200 .
op fy : Int -> Int .
op fx : Int -> Int .
eq fy(a) = a - y  .
eq fx(a) = 2 * a  .
ceq f(a) = fx(fy(a)) if a > 1000 .
ceq f(a) = fx(a) if a < 1000 or a == 1000 .
endfm
red f(1500) .
red f(900) .
```
(M2)

3.2.2. if else statement

```
int f(int a, int b)
{int x = 10;
if a > b  x = (2*a) + x;
else x = (5*b) + x;
return x;}

return y;}
```
(C3)

```
fmod Cfunction is
protecting INT .
op f : Int Int  -> Int .
vars a b : Int .
op x :  -> Int [ctor] .
eq x = 10 .
op fx1 : Int Int -> Int .
op fx2 : Int Int -> Int .
eq fx1(a,b) = 2 * a + x  .
eq fx2(a,b) = 5 * b + x  .
ceq f(a,b) = fx1(a,b) if a > b .
ceq f(a,b) = fx2(a,b) if a < b or a == b
.
endfm
red f(8, 6) .
red f(6, 8) .
```
(M3)

Here, for each branch of the if statement, a conditional equation is created.

DOI: doi.org/10.35970/jinita.v5i1.1846

3.2.3. Nested if

All assignments are identified for which we define for each assignment a conditional equation. The condition of each equation is a conjunction (i.e. and) between all nested conditions. An unconditional equation with the attribute 'owise' is added to specify the remaining cases.

```
int f(int a, int b)
{int x = 0;
if a > b
   if a < 10  x = a-b;
   else x = a - (2*b);
else
 if a < b
  if b < 10  x = a+b;
   else x = 2*a;
return x;}
            (C4)
```

```
fmod Cfunction is
protecting INT .
op f : Int Int  -> Int .
vars a b : Int .
op x :  -> Int [ctor] .
eq x = 0 .
op fx1 : Int Int -> Int .
op fx2 : Int Int -> Int .
op fx3 : Int Int -> Int .
op fx4 : Int -> Int .
eq fx1(a,b) = a - b .
eq fx2(a,b) = a - (2 * b) .
eq fx3(a,b) = a + b .
eq fx4(a) = 2 * a .
ceq f(a,b) = fx1(a,b) if a > b and a < 10 .
ceq f(a,b) = fx2(a,b) if a > b and (a == 10 or a >
10) .
ceq f(a,b) = fx3(a,b) if a < b and b < 10 .
ceq f(a,b) = fx4(a) if a < b and (b == 10 or b >
10) .
eq f(a,b) = x [owise] .
endfm
red f(9, 8) .
red f(12, 8) .
red f(8, 9) .
red f(8, 12) .
                          (M4)
```

3.2.4. while loop

In this paper, we will consider the basic while loop since other C loops as 'for loop' can be transformed to the *while* loop. Iterative loops can be transformed to recursive functions [9]. To explain, how we can transform a while loop to a Maude functional module, we consider the C function that calculate the sum of N first integers. The general idea is to transform the loop statement into a recursive conditional equation called loop. The signature of the loop operation has to contain all parameters that figure in the condition of the while and all variables occurring in the left side of all assignments defined inside the body of the while loop. Then the recursive call is established while modifying the values of the variables. For each loop equation, we have to define one conditional recursive equation and one non-recursive one to terminate the recursion. Initial values of local variables are transmitted to the loop equation from the main equation. From code C5, we can derive the loop equation having the following signature: op loop : Int Int Int -> Int .This signature has three arguments corresponding to n, i and som variables. Each of them is an integer. The sort of the returned value is the same of the type of the returned value by the C function (in this case, Int).

```
int sum(int n)
{int i = 1; int som = 0;
While (i <= n) {
som = som + i;
i = i + 1;}
return som;}
          (C5)
```

```
fmod SUM is
protecting INT .
op sum : Int  -> Int .
op loop : Int Int Int  -> Int .
vars i n som : Int .
ceq loop(n, som, i) = loop(n, som + i, i + 1) if i < n
or i == n .
ceq loop(n, som, i) = som if i > n .
eq sum(n) = loop(n, 0, 1) .
endfm
red sum(10, 0, 1) .
                          (M5)
```

3.2.5. while loop with break

Listing C6 shows a C function that checks whether an integer number is prime or not. In this example, the statement break is used inside the while loop and forces the exit of the loop even the loop condition is true. The idea of transformation is similar to the first case, but here, we have to add a conditional equation reflecting the exit of the loop due to the break statement.

DOI: doi.org/10.35970/jinita.v5i1.1846

```
int prime(int n)
{int i = 2;
While (i <= n / 2) {
if n % i = 0 break;
i = i + 1 ;}
if i > n / 2 return 1 else
if (n % i = 0) return 0;}
            (C6)
```

```
fmod BREAK is
protecting INT .
op prime : Int  -> Bool .
op loop : Int Int  -> Bool .
vars i n som : Int .
ceq loop(n, i) = loop(n, i + 1) if (i < n quo 2 or
i == n quo 2) and (n rem i =/= 0) .
ceq loop(n, i) = true if i > n quo 2 .
ceq loop(n, i) = false if n rem i == 0 .
eq prime(n) = loop(n, 2) .
endfm
red prime(117) .
red prime(223) .
                        (M6)
```

3.2.6. while loop with continue

Continue statement forces the next iteration of the loop to take place skipping any code in between. Continue can be replaced by an if-else statement, where the skipped code (i.e. all the code just after continue) is put in the else branch as shown in code C'7. Here we have to define three loop conditional equations: one for the if branch, one for the else branch and the third for loop exit.

```
int f(int n){
int a=1;int s=0;
while a < n{
if a == 7{
a=a+1;
continue;}
s=s+a;
a=a+1;}
return s ;
           (C7)
```

```
int f(int n){
int a=1;int s=0;
while a < n{
if a == 7
a=a+1; else{
s=s+a;
a=a+1;}}
return s;
           (C'7)
```

```
fmod CONTINUE is
protecting INT .
op f : Int -> Int .
op loop : Int Int Int -> Int .
vars n a s : Int .
ceq loop(n,a,s) = loop(n, a + 1, s) if a < n and a == 7 .
ceq loop(n,a,s) = loop(n, a + 1, s + a) if a < n and a =/= 7 .
ceq loop(n,a,s) = s if a > n or a == n .
eq f(n) = loop(n, 1, 0) .
endfm
red f(10) .
                                    (M7)
```

3.2.7. Nested while

We start with the most inner while loop by transforming it to a conditional equation named loop*i*, where i gives the maximum level of loops nesting following the technique described above. Then, we transform the less inner while loop by defining a second conditional equation loop*i-1* and so on until we reach the most outer loop. Each inner loop equation becomes an argument of the next outer loop equation. The signature of the most outer loop has to include all the arguments that figure in the embodied while loops conditions and all variables that occur in the left right of assignments of the while loops bodies. Finally, the equation of the main function is defined in term of the most outer loop equation by transmitting the necessary parameters and initial values. M8 shows the Maude code for the C8 code. In this example, we have two nested while loops. The most inner loop is transformed to the conditional equation l*oop2* with three arguments, however, the most outer loop is transformed to l*oop1* with five parameters. loop2 becomes an argument of loop1.

DOI: doi.org/10.35970/jinita.v5i1.1846

```
int f(int n, int m)
{int i = 1; int x = 0;
While (i <= n ) {
int j = 1;
while (j <= m) {
x = x + 2;
j = j + 1 ;
}
i = i + 1 ;}
return x;}
          (C8)
```

```
fmod Cfunction is
protecting INT .
op f : Int Int  -> Int .
op loop2 : Int Int Int -> Int .
op loop1 : Int Int Int Int Int -> Int .
vars i j n m x : Int .
ceq loop2(m, x, j) = loop2(m, x + 2, j + 1) if j < m or j
== m .
ceq loop2(m, x, j) = x if j > m .
ceq loop1(n, m, x, j, i) =  loop1(n, m, loop2(m, x, j),
j, i + 1)   if i < n or i == n .
ceq loop1(n, m, x, j, i) =  x if i > n .
eq f(n, m) = loop1(n, m, 0, 1, 1) .
endfm
red f(10, 5) .
                              (M8)
```

4. Functions without parameters

A C function can take no parameters. In C, *int f()* means that f returns an integer value while it can take any number of arguments. In this case, the compiler will not perform any checking on the number and types on parameters when calling this function. If we want no arguments then we have to declare as *int f(void)*. In Maude, an operation without parameters is considered as a constant function declared without specifying the function definition domain. Many cases of C functions without parameters exist:

4.1. A function, which computes a constant value

Let's consider the function f() returning the value of Pi = 3.14.

```
long f(void){
return 3.14;}
        (C9)
```

```
op f : -> Float .
eq f = 3.14 .
          (M9)
```

4.2. A function, which calculates a certain value using some local variables

```
int f(void){
int x = 100;
int y = 5;
return x + y;}
        (C10)
```

```
op f : -> Int .
op x : -> Int .
op y : -> Int .
eq x = 100 .
eq y = 5 .
eq f = x + y .
       (M10)
```

4.3. A function, which calculates some value using some local variables whose values are introduced by the user

In order to transform the C function to a Maude code, all local variables introduced by the user (i.e. via scanf) will be considered as parameters. In C code C11, the two variables x and y will be added to the signature of the Maude operation.

```
#include <stdio.h>
int f(void){
int x;
int y;
scanf("%d", &x);
scanf("%d", &y);
return x + y;}
        (C11)
```

```
op f : Int Int -> Int .
vars x y : Int .
eq f(x, y) = x + y .
                (M11)
```

4.4. A function, which has access to a global variable

A pure functional language does not manipulate global variables. A global variable can be a source of side effects in imperative languages such as C because it is visible to all functions and each function can read and/or update its value. In order to transform a C function to Maude, each global variable will be considered as a parameter of this function, but also should be considered as a return value for the same function, so any updating in the global variable value can be visible to the other functions. C12 shows an

DOI: doi.org/10.35970/jinita.v5i1.1846

example of a function that updates the value of a global variable called y, then use this value to compute a certain value z. The Maude code for this function is showed in M12. Operation *f* has now one parameter y and two returned values: z and y. Here, two solutions are possible: whether we specify the rang of the operation as a list of pairs if the sort of the global variable is same as the sort of the returned value of the function. Each pair is composed of the name of the variable with its value. If the sorts of the returned value and the global variable are different, then the rang of the function can be declared as a *struct* sort.
Let us assume that y is also an integer, so we can puts both the return value and y in the same list. In this example, the list is composed of two pairs: *("f", z(y(val), x))* and *(g, y(val))* where *"f"* is the name of the operation (i.e C function) and *g* designates the global variable. *z(y(val), x)* is the expression calculating the return value of *f()*, and *y(val)* is the new value of the global variable whose initial value when reading by the function is *val*.

```
int f(void){
int x = 20;
y = y + 1000;
int z = y + x;
return z ;}
        (C12)
```

```
sort List .
sort pair .
op _,_ : String Int -> pair .
subsort pair < List .
op nil : -> List [ctor] .
op _;_ : List List -> List [ctor assoc id: nil ] .
op f : List -> List .
op x : -> Int .
op y : Int -> Int .
vars val a b : Int .
var g : String .
op z : Int Int -> Int .
eq x = 20 .
eq y(val) = val + 1000 .
eq z(a, b) = a + b .
eq f(g, val) = ("f", z(y(val), x)) ; (g, y(val)) .
endfm
red f("y", 600) .
                          (M12)
```

5. Transformation of Functions without returned values (procedures)

In C, a function without a return value is declared as *void*, In this case the return statement is usually omitted. In general, a procedure has side effect statements and accepts parameters passing by reference declared as pointers. Since C procedures show nondeterministic behavior, we will transform them as Maude system modules. Rewiring logic match well with nondeterministic behaviors.
In contrast to equational logic, Maude rewriting logic can be used to specify the operational semantics of the imperative code. Thus, it specifies the effect of the execution of each statement on the state of the memory code. Each variable inside the procedure will be specified by a triplet including its name, its type and its current value as follows:

```
sorts state var type value .
subsort var < String .
subsort value < Int .
op undef : -> value .
op nil : -> state [ctor] .
ops int intlong char string double : -> type [ctor] .
op <_,_,_> : var type value -> state .
op __ : state state -> state [assoc comm id:nil ] .
```

Here, we define the concatenation of states (i.e. multiset of states) as associative, commutative with identity. The constant *nil* designates a null instruction (i.e. the ';' statement) which has no effect on the state of the code. Each triplet is specified as <_,_,_>. We add two constants START and END to specify the start and the end of the C code. The constant *Undef* specifies any undefined (i.e. uninitialized) value. We put all the above definitions in a functional module called PROCEDURE.

DOI: doi.org/10.35970/jinita.v5i1.1846

```
fmod PROCEDURE is
protecting INT .
protecting FLOAT .
protecting STRING .
sorts state var type value .
subsort var < String .
subsort value < Int .
op undef : -> value .
op nil : -> state [ctor] .
ops int intlong char string double : -> type [ctor]
.
ops START END : -> state .
op <_,_,_> : var type value -> state .
op __ : state state -> state [assoc comm id: nil ] .
endfm
```

More generally, each C statement modifying the state of the memory can be specified as a rewriting rule of the form: t -> t', where t and t' are terms designating the state before and after the execution of the C statement. Terms t and t' can include sub-terms specifying many C statements. For example, let us assume that the C code has three assignments:
int i =10; int x =20; int y =i+x;

Before these assignments, suppose that each of the three variables takes the *undef* value. Remark that the first two assignments are independent. The third assignment however depends on the two previous statements. A possible Maude code for this is:
rl [init] : <"i", int, undef> <"x", int, undef><"y", int, undef>=><"i", int, 10> <"x ", int, 20><"y", int, undef> .
rl [inity] : <"i", int, i> <"x", int, x><"y", int, undef>=><"i", int, i> <"x", int, x><"y", int, i + x> .

The first rule, which is labelled [init], permits the initialization of the two variables i and x. Variable y remains uninitialized. The second rule [inity] permits the initialization of the variable y based on values i and x. This rule also maintains the states of i and x. Nevertheless, if we suppose that variables i and x will not be never used later, the second rule becomes:
rl [inity] : <"i", int, i> <"x", int, x><"y", int, undef>=><"y", int, i + x> .
It is important to notice that rewriting logic has a concurrent semantic, the imperative paradigm semantic however has a sequential semantic, so in order to specify C sequentiality, we add to the right term of a rewriting rule and to the left term of its next rewriting rule, a sort of tag. By this, a rule starts the rewriting only if its previous rule finishes the rewriting.
For instance, if we want the two rules [init] and [inity] will be executed sequentially, we can define a tag called E1 as follows:
op E1 : -> state [ctor] .
rl [init] : <"i", int, undef> <"x", int, undef><"y", int, undef>=><"i", int, 10> <"x", int, 20><"y", int, undef> E1 .
rl [inity] : E1 <"i", int, i> <"x", int, x><"y", int, undef>=><"i", int, i> <"x", int, x><"y", int, i + x> .
We say that the first rule produces E1 and the second rule consumes E1.
The position of E1 does not matter since the operation of concatenation is commutative.
Finally, the sub-term <"y", int, undef> in the first rule occurs in the right and in the left of the rule. Because rewriting logic is the logic of changing, this sub-term can be removed from the first rule, but it must figure in the initial configuration (initial state) as a parameter of the command *rew*. So, the rule [init] can be rewritten as:
rl [init] : <"i", int, undef> <"x", int, undef> => <"i", int, 10> <"x", int, 20> E1 .
Of course, there are other sophisticated solutions to specify sequentiality. For example by resorting to Maude meta-level to define a sequential strategy.

DOI: doi.org/10.35970/jinita.v5i1.1846

Conditional statements and loops are specified using conditional rewriting rules. For instance, code C14 is transformed to Maude code M14. Here, the loop while is specified using two conditional rewriting rules: [while] and [endWhile].

```
#include <stdio.h>
void sum(int n){
int i=1; int s=0;
while i <= n {
s=s+i;
i=i+1;
}
printf(s);}
          (C14)
```

```
mod sum is
protecting PROCEDURE .
op E1 : -> state .
vars n i s : Int .
rl[start] : START < "n", int, n > =>
 < "n", int, n > < "i", int, 1 > < "s", int, 0 > E1
.
crl[while] : E1 < "n", int, n > < "s", int, s > <
"i", int, i > => E1 < "n", int, n > < "s", int, s +
i > < "i", int, i + 1 > if i < n or i == n .
crl[endwhile] : E1 < "n", int, n > < "s", int, s >
< "i", int, i > => END if i > n [print s] .
endm
set print attribute on .
rew START < "n", int, 10 > .
                          (M14)
```

Rewriting logic is suitable for specifying side effect statements, which show nondeterministic behavior. In essence, rewriting rules execute concurrently but since the Maude interpreter is sequential, concurrent transitions are simulated by corresponding interleavings of sequential rewriting steps.
An example of a C side effect is:
int x, y = (x = 3) + (x = 4);
printf("x = %d y =%d \n", x, y);
Possible outputs can be:
x=4 y=7,
x=3 y=7,
x=4 y=8 (if compiled with gcc -03)
The above C code can be specified as a set of concurrent rewriting rules as follows:
rl [0] : < "x", int, undef > => < "x", int, 3 > .
rl [1] : < "x", int, undef > => < "x", int, 4 > .
rl [2] : < "x", int, x > < "y", int, undef > => < "y", int, x + x > [print x y] .
The assignment y =x++ can be specified as:
rl [0] : < "y", int, undef > < "x", int, x > => < "y", int, x >
< "x", int, x + 1 > .
The assignment y=++x can be specified as:
rl [1] : < "y", int, undef > < "x", int, x > => < "y", int, x + 1 > < "x", int, x + 1 > .

In a general fashion, a function with a returned value can be considered as a special case of a procedure. Thus any C function with returned value can be transformed to its equivalent procedure by adding the returned value of the function into the procedure parameters list and remove the returned statement from the function code . This new parameter should be passed by reference. As an example, let us assume the C code C15. Here, the function sum can be transformed to a procedure of the same name with two parameters.

```
int sum(int n)
{int i = 1; int som = 0;
While (i <= n) {
som = som + i;
i = i + 1;}
return som;}
          (C15)
```

```
void sum(int n, int* som)
{int i = 1; som = 0;
While (i <= n) {
som = som + i;
i = i + 1;}}
                (C15')
```

This transformation from a function to a procedure, gives us the idea to transform Maude equations to rewriting rules and vice versa. An equation *eq t = t'* can be interpreted as *t -> t'* and *t' -> t*. for the purpose of transformation from a function to a procedure, we will take the rule *t -> t'* where the term t is an operation signature and t' is either an operation signature or a constant. The same thing with a conditional equation. So a conditional equation of the form

DOI: doi.org/10.35970/jinita.v5i1.1846

ceq t = t' if C can be replaced by *crl [R] : t -> t' if C* . To explain, how the transformation is done, we take the code M5 as an example. Here, we have two operations *sum* and *loop*. Loop is a recursive , but sum is not. ceq loop(n, som, i) = loop(n, som + i, i + 1) if i < n or i == n . is transformed to:
crl [loop1] : < "n", int, n > < "som", int, som > < "i", int, i > => < "n", int, n > < "som", int, som + i > < "i", int, i + 1 > if i < n or i == n .
ceq loop(n, som, i) = som if i > n . is transformed to:
crl [loop2] : < "n", int, n > < "som", int, som > < "i", int, i > => < "som", int, som > if i > n .
eq sum(n) = loop(n, 0, 1) . is transformed to:
rl [sum] : < "n", int, n > => < "n", int, n > < "som", int, 0 > < "i", int, 1 > .

For each functional module, we create a system module named *PROCEDUREFunctionalModuleName*. The functional module will be imported to recuperate variables declarations. However, all operations signatures and equations have to be removed from the functional module. By adding *** (three asterisks), the corresponding definition becomes a commentary.

```
fmod SUM' is
***protecting INT .
***op sum : Int  -> Int .
***op loop : Int Int Int  -> Int .
vars i n som : Int .
***ceq loop(n, som, i) = loop(n, som + i,
i + 1) if i < n or i == n .
***ceq loop(n, som, i) = som if i > n .
***eq sum(n) = loop(n, 0, 1) .
endfm
***red sum(10, 0, 1) .
                  (M'5)
```

```
mod PROCEDURESUM is
protecting PROCEDURE .
protecting SUM' .
crl [loop1] : < "n", int, n > < "som",
int, som > < "i", int, i > => < "n",
int, n > < "som", int, som + i > < "i",
int, i + 1 > if i < n or i == n .
crl [loop2] : < "n", int, n > < "som",
int, som > < "i", int, i > => < "som",
int, som > if i > n .
rl [sum] : < "n", int, n > => < "n",
int, n > < "som", int, 0 > < "i", int, 1
> .
endm
rew < "n", int, 10 > .
                  (M''5)
```

6. Verification of Maude specifications

6.1. The 'reduce' command

With the reduce command; we can evaluate the functional correction of Maude equations. As examples, we show the reduction of M6 and M8 using core Maude interpreter 2.6.

Figure 1. Result of reduction of functional module M6

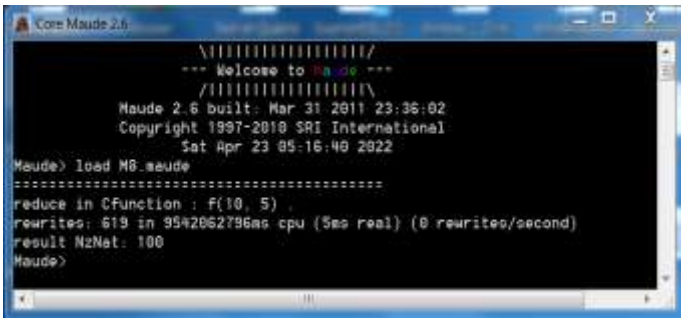

Figure 2. Result of reduction of functional module M8

6.2. Simulation using the command 'rewrite'

As example, we show the rewriting of system module M''5.

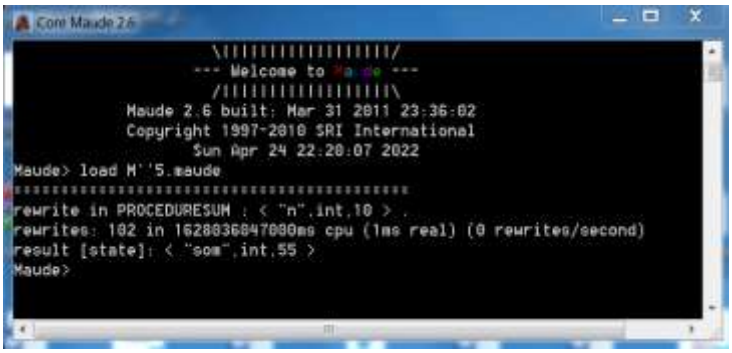

Figure 3. The rewriting result of the system module M''5

6.3. Reachability analysis using the command "search"

As an example, we want to check whether the state ERROR in (C16) can be reached or not. Code M16 shows the Maude specification of the corresponding C code with the application of the *search* command.

```
void test(int n){
int i = 0; int j = 0
while (i < 2n) {
if (i < n) i = i+1;
else j = i+1;
i = i+1}
if i < j printf
("error");}
        (C16)
```

```
mod TEST is
protecting PROCEDURE .
op ERROR : -> state .
rl [start] : START < "i", int, undef > < "j", int, undef >
=> < "i", int, 0 > < "j", int, 0 > .
crl [loop1] : < "n", int, n > < "i", int, i > => < "n",
int, n > < "i", int, i + 2 > if ( i < 2 * n ) and ( i < n )
.
crl [loop2] : < "n", int, n > < "i", int, i > < "j", int, j
> => < "n", int, n > < "j", int, i + 1 > < "i", int, i + 1
> if ( i < 2 * n ) and ( i > n or i == n) .
crl [endloop] : < "n", int, n > < "i", int, i > < "j", int,
j > => END if ( i > 2 * n or i == 2 * n ) .
crl [error] : < "n", int, n > < "i", int, i > < "j", int, j
> => ERROR if ( i > 2 * n or i == 2 * n ) and ( i < j )
[print "error" ] .
endm
search START < "n", int, n > < "i", int, undef > < "j",
int, undef > =>* ERROR .
                              (M16)
```

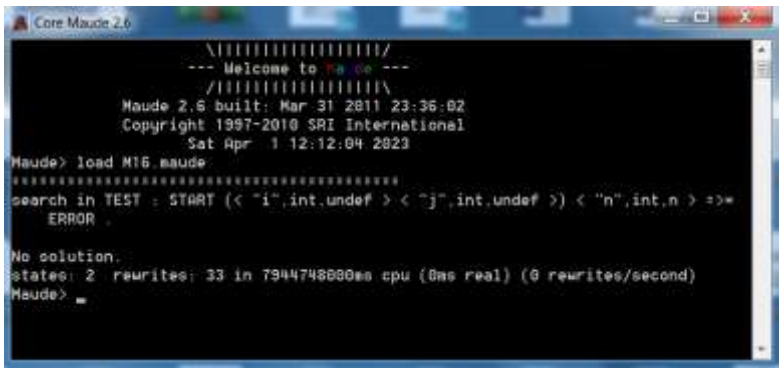

Figure 4. The reachability analysis result of M16

7. Conclusion and future work

In this paper, we showed how C functions could be transformed to Maude functional and system modules. Maude functional modules implement equational theory. System modules, however implement rewriting theory. This transformation will enable formal verification of C programs via available Maude tools. Our future investigations will focus on the automation of the proposed transformation while taking into account other C constructors and features as pointers, complex types like structures, unions, arrays and casting.

REFERENCES

[1] D. Beyer, "Software Verification: 10th Comparative Evaluation (SV-COMP 2021)," Groote, J.F., Larsen, K.G. (eds) Tools and Algorithms for the Construction and Analysis of Systems. TACAS 2021. Lecture Notes in Computer Science (vol. 12652). Springer, 2021, doi: 10.1007/978-3-030-72013-1_24.

[2] M. Bonger, *Reasoning about C programs*, PhD thesis, University of Queensland. 1998.

[3] F. Boutekkouk, "Maude Specification Generation from VHDL", Das, V.V., Chaba, Y. (eds) Mobile Communication and Power Engineering. AIM 2012. Communications in Computer and Information Science 296, Springer, 2013, doi: 10.1007/978-3-642-35864-7_56

[4] M. Clavel, F. Duran, S. Eker, S. Escobar, P. Lincoln, N. Marti-Oliet, J. Meseguer, and C.Talcott, *Maude Manual (Version 2.7.1)*, 2016.

[5] P. Cuoq, F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski, "Frama-C A Software Analysis Perspective," Formal Aspects of Computing, 2012.

[6] C.M. Ellison, *A Formal Semantics of C with Applications*. PhD thesis, University of Illinois, 2012.

[7] J.A. Goguen and G. Malcolm, *Algebraic Semantics of Imperative Programs* (Book), MIT Press, ISBN: 9780262071727, 1996.

[8] P. Hartel and H. Muller, *Functional C*. Revision 6.8. 1999.

[9] D. Insa and J. Silva, "Automatic Transformation of Iterative Loops into Recursive Methods," n: CoRR abs/1410.4956, 2014.

[10] ISO/IEC 9899:2018, Information technology — Programming languages — C, https://www.iso.org/standard/74528.html

[11] F. Ivancic, I. Shlyakhter, A. Gupta, M.K. Ganai, V. Kahlon, C. Wang and Z. Yang, "Model Checking C Programs Using F-SOFT," International Conference on Computer Design 31 October, San Jose, CA, USA, 2005.

[12] K. Jiang, *Model Checking C Programs by Translating C to Promela*, Master thesis, Linkoping University, Sweden, 2009.

[13] R.J. Krebbers, *The C standard formalized in Coq*, PhD thesis, Radboud University Nijmegen, 2015.

[14] J.B. MacLennan, *Functional Programming: Practice and Theory*. Addison-Wesley, 1990.

[15] M. Norrish, *C Formalised in HOL*. PhD thesis, University of Cambridge, 1998.

[16] M. Sammler, R. Lepigre, and R. Krebbers, "RefjnedC: Automating the Foundational Verifjcation of C Code with Refjned Ownership Types," PLDI '21, Canada, 2021.

[17] N. Schirmer, *Verification of Sequential Imperative Programs in Isabelle/HOL*. PhD thesis, Technische Universitat Munchen, 2005.

[18] A. Stefanescu, "MatchC: A Matching Logic Reachability Verifier Using the K Framework," Electronic Notes in Theoretical Computer Science 304, pp. 183–198, 2014.

[19] Summary of C/C++ program verification tools - Programmer Sought, 20/07/2021 https://programmersought.com/article/90174848682/

[20] A. Verdejo and N. Martı-Oliet, "Executable Structural Operational Semantics in Maude," The Journal of Logic and Algebraic Programming, 67, pp. 226–293, 2006.