C2TLA+: A translator from C to TLA+

Amira METHNI (CEA/CNAM)
Matthieu LEMERRE (CEA) & Belgacem BEN HEDIA (CEA)
Serge HADDAD (ENS Cachan) & Kamel BARKAOUI (CNAM)

June 3, 2014
Outline

- Introduction
- General approach
- Translation from C to TLA+
  - Memory model
  - Expressions
  - Intra-procedural control flow
  - Inter-procedural control flow
  - Generating specification
  - Examples of properties
- Conclusion
Introduction

Context

- C is a low level language
- Programs are concurrent
  - Verifying C code is challenging (presence of pointer, pointer arithmetic's...)

Motivation

- Verifying an implementation model.
- Guaranteeing the absence of certain classes of errors.

Method

- Automatically translate a TLA+ specification from input C codes.
- Using automated tools to verify concurrent C programs against a set of safety and liveness properties.
General approach

C files → CIL → C2TLA+ → TLA+ specification → TLC Model checker

- TLA+ properties

- Property violated?
  - yes → Error trace
  - No → OK

TLA+ specification

TLA+ properties

C2TLA+
**General approach**

CIL (C Intermediate Language) is a high-level representation along with a set of tools that permit easy analysis and source-to-source transformation of C programs.

Some of CIL’s simplifications:

- All forms of loops (while, for and do) are compiled internally as a single while(1) looping construct with explicit goto statements (for termination.)

```c
while (x<10) {
    if (x == 8)
        continue;
    x++;
}
```

```c
while (1) {
    while_continue: /* internal */ ;
    if (! (x < 10))
        goto while_break;
    if (x == 8)
        goto while_continue;
    x ++;
}
while_break: /* internal */ ;
```
**CIL (C Intermediate Language)** is a high-level representation along with a set of tools that permit easy analysis and source-to-source transformation of C programs.

- Some of CIL’s simplifications:
  - Expressions that contain side-effects are separated into statements.

```c
C

return(y ++ + f(x++));
```

```c
CIL

int tmp = y;
y ++;
int tmp_0 = x;
x ++;
int tmp_1 = f(tmp_0);
int __retres = tmp + tmp_1;
return (__retres);
```
General approach

C files → **CIL** → **C2TLA+** → TLC Model checker

TLA+ specification → TLA+ properties

- Property violated?
  - Yes → Error trace
  - No → OK
Expressions: pointers, pointer arithmetic, referencing, dereferencing (&), array indexing, structure members (.), arithmetic (*, +, -, %, /), relational (>, >=, <, <=, ==, !=) and logical (&&, ||, !) operators;

Statements: assignment, conditions (if, if/else), loops (for/do-while/while), goto, break, continue, return;

Data types: integers (int), structures (struct), enumerations (enum);

Value-returning function of int or pointer type;

Recursion.

C2TLA does not support: functions pointer, dynamic memory allocation and assignments of structures types.
General approach

- Using TLC to verify properties
  - Safety
    - Problems because of pointers and arrays (dereferencing null pointer).
    - Invariants over variables values.
    - Mutual exclusion.
  - Liveness
    - Termination.
    - Starvation-freedom (each waiting process will eventually enter its critical section).
Concurrent program consists in several interleaved sequences of operations called processes (corresponding to threads in C).

C2TLA+ attributes a unique identifier to each process, and defines the constant Procs to be the set of all process identifiers.
Four memory regions:

1) A region that contains global (and static) variables, called `mem`.

- Shared by processes.
- Modeled by an array (function).

```c
int x = 0;
int y = 2;
int z;
int max(int a, int b)
{ if (a>=b)
    return a;
  else return b; }

int process0()
{ int i = 1;
  x = x + i;
  y = max(x, y);
  x = y + 1;
  return x; }

void process1()
{ int j = 0;
  x = max(x, y); }
```
Four memory regions:

2) A region contains local variables and function parameters, called `stack_data`.

- This region is modeled by an array of sequences and is composed of stack frames.
- Each stack frame corresponds to a call to a function which has not yet terminated with a return.

```
1- int x = 0;
2- int y = 2;
3- int z;
4-
5- int max(int a, int b)
6- { if (a>=b)
7-     return a;
8-   else return b; }
9-
10- int process0()
11- {
12-   int i = 1;
13-   x = x + i;
14-   y = max(x, y);
15-   x = y + 1;
16-   return x; }
17- void process1()
18- {
19-   int j = 0;
20-   x = max(x, y); }
```
Four memory regions:

2) A region contains local variables and function parameters, called *stack_data*.

- This region is modeled by an array of sequences and is composed of stack frames.
- Each stack frame corresponds to a call to a function which has not yet terminated with a return.

![Stack Data Diagram]

```c
int x = 0;
int y = 2;
int z;

int max(int a, int b)
{
    if (a >= b)
        return a;
    else return b;
}

int process0()
{
    int i = 1;
    x = x + i;
    y = max(x, y);
    x = y + 1;
    return x;
}

void process1()
{
    int j = 0;
    x = max(x, y);
}
```

This region is modeled by an array of sequences and is composed of stack frames. Each stack frame corresponds to a call to a function which has not yet terminated with a return.
Memory Model

Four memory regions:

3) A region that stores the program counter of each process \((stack\_regs)\).

- It associates to each process a stack of records.
- Each record contains two fields:
  - \(pc\), the program counter, represented by a tuple function \(<\text{name, label}>\) (Labels values are given by CIL).
  - \(fp\), the frame pointer, contains the base offset of the current stack frame.

```
int x = 0;
int y = 2;
int z;

int max(int a, int b)
{
    if (a>=b)
        return a;
    else return b;
}

int process0()
{
    int i = 1;
    x = x + i;
    y = max(x, y);
    x = y + 1;
    return x;
}

void process1()
{
    int j = 0;
    x = max(x, y);
}
```

```
1- int x = 0;
2- int y = 2;
3- int z;
4- int max(int a, int b)
6- {if (a>=b)
7-     return a;
8- else return b;}
9- int process0()
10- {int i = 1;
12-     x = x + i;
13-     y = max(x, y);
14-     x = y + 1;
15-     return x; }
16- void process1()
17- {int j = 0;
18-     x = max(x, y); }
```
Four memory regions:

3) A region that stores the program counter of each process (stack_regs).

It associates to each process a stack of records.

Each record contains two fields:

- pc, the program counter, represented by a tuple function <<name, label>> (Labels values are given by CIL).
- fp, the frame pointer, contains the base offset of the current stack frame.

```
1- int x = 0;
2- int y = 2;
3- int z;
4-
5- int max(int a, int b)
6- { if (a>=b)
7-     return a;
8-   else return b;}
9-
10- int process0()
11- { int i = 1;
12-     x = x + i;
13-     y = max(x, y);
14-     x = y + 1;
15-     return x; }
16-
17- void process1()
18- { int j = 0;
19-     x = max(x, y); }
```
Memory Model

Four memory regions:

3) A region that stores the program counter of each process (stack_regs).

- It associates to each process a stack of records.
- Each record contains two fields:
  - \(pc\), the program counter, represented by a tuple function \(<name, label>\> (Labels values are given by CIL).
  - \(fp\), the frame pointer, contains the base offset of the current stack frame.

```
int x = 0;
int y = 2;
int z;
int max(int a, int b)
{
  if (a>=b)
    return a;
  else return b;
}
int process0()
{
  int i = 1;
  x = x + i;
  y = max(x, y);
  x = y + 1;
  return x;
}
void process1()
{
  int j = 0;
  x = max(x, y);
}
```
Four memory regions:

3) A region that stores the program counter of each process (stack_regs).

- It associates to each process a stack of records.
- Each record contains two fields:
  - \( pc \), the program counter, represented by a tuple function \(<\text{name}, \text{label}>\) (Labels values are given by CIL).
  - \( fp \), the frame pointer, contains the base offset of the current stack frame.

\[
\begin{align*}
\text{mem} & \quad \downarrow \quad \uparrow \\
\text{stack_data}[0] & \\
& \quad \downarrow \quad \uparrow \\
& \quad \downarrow \quad \uparrow \\
\text{stack_regs}[0] & \\
\end{align*}
\]

1. \( \text{int} \ x = 0; \)
2. \( \text{int} \ y = 2; \)
3. \( \text{int} \ z; \)
4. \( \text{int} \ \text{max}(\text{int} \ a, \text{int} \ b) \)
   \( \{ \text{if} \ (a>=b) \) \\
   \quad \text{return} \ a; \)
   \( \quad \text{else} \ \text{return} \ b; \} \)
5. \( \text{int} \ \text{process0}(); \)
   \( \quad \text{int} \ i = 1; \)
   \( \quad x = x + i; \)
   \( \quad y = \text{max}(x, y); \)
   \( \quad x = y + 1; \)
   \( \quad \text{return} \ x; \) \}
6. \( \text{void} \ \text{process1}(); \)
   \( \quad \text{int} \ j = 0; \)
   \( \quad x = \text{max}(x, y); \) 

The memory model consists of four memory regions:

- A region that stores the program counter of each process (stack_regs).
- It associates to each process a stack of records.
- Each record contains two fields:
  - \( pc \), the program counter, represented by a tuple function \(<\text{name}, \text{label}>\) (Labels values are given by CIL).
  - \( fp \), the frame pointer, contains the base offset of the current stack frame.
Four memory regions:

4) A region contains the values returned by processes, called \textit{ret}.

- This region is modeled by an array and indexed by process identifier.

```
1- int x = 0;
2- int y = 2;
3- int z;
4- int max(int a, int b)
   {if (a >= b)
      return a;
   else return b;}
7- return x;
9- void process1()
   {int j = 0;
    x = max(x, y);
    return x;
   }
```
C2TLA+ maps each C variable to unique TLA+ constant (address) modeled by a record with two fields:

- **loc**: memory region (mem or stack_data).
- **off**: offset in the considered memory region.

Example

\[
\text{Addr}_x \overset{\Delta}{=} [\text{loc} \mapsto "mem", \text{offs} \mapsto 0]
\]

\[
\text{Addr}_{\text{process0}_i} \overset{\Delta}{=} [\text{loc} \mapsto "stack\_data", \text{offs} \mapsto 0]
\]

```plaintext
int x = 0;
int y = 2;
int z;

int process0(){
    int i = 1;
    x = x + i;
    y = max(x,y);
    x = y + 1;
    return x;
}

void process1(){
    int j = 0;
    x = max(x,y);
    x = max(x,y);
}
```

Example table:

<table>
<thead>
<tr>
<th>mem</th>
<th>stack_data[0]</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>[val -&gt; 1]</td>
</tr>
<tr>
<td>y</td>
<td>[val -&gt; 2]</td>
</tr>
<tr>
<td>z</td>
<td>[val -&gt; Undef]</td>
</tr>
</tbody>
</table>

The `max` stack frame is represented by `[val -> 2]` in `stack_data[0]`. The `process0` stack frame is represented by `[val -> 1]` in `stack_data[0]`.
C2TLA+ maps each C variable to unique TLA+ constant modeled by a record with two fields:

- **loc**: memory region *(mem or stack_data)*.
- **off**: offset in the considered memory region.

**Loading operation**

- A *lvalue* is evaluated to an address and which refers to a region of storage.
- Accessing the value stored in this region is performed using the TLA+ operator *load()*.  

\[
\text{load}(id, \text{ptr}) \triangleq \begin{cases} 
\text{IF } \text{ptr.loc} = \text{"mem" THEN } \text{mem}[\text{ptr.off}] \\
\text{ELSE } \text{stack_data}[id][\text{Head(stack_regs[id]).fp + ptr.off}] 
\end{cases}
\]
C2TLA+ maps each C variable to unique TLA+ constant modeled by a record with two fields:

- \textit{loc} : memory region (\textit{mem} ou \textit{stack\_data}).
- \textit{off} : offset in the considered memory region.

### Loading operation

### Assignment operation

\[\text{store}(id, ptr, value) \triangleq\]
\[\land \land ptr.loc = \text{“mem”}\]
\[\land mem’ = [mem \text{ EXCEPT } ![ptr.offs] = value]\]
\[\land \text{UNCHANGED stack\_data}\]
\[\land \land ptr.loc = \text{“stack\_data”}\]
\[\land stack\_data’ = [stack\_data \text{ EXCEPT } ![id][\text{Head}(stack\_regs[id]).fp + ptr.offs] = value]\]
\[\land \text{UNCHANGED mem}\]

#### Example

The statement \(i = 1;\) is translated into TLA+ as \(\text{store}(id, \text{Addr\_process0\_i}, [val \mapsto 1])\)
Expressions

- **Arrays**
  - Accessing an array element in C2TLA+ requires computing the offset using the size of the elements, the index and the base address of the array.
  - Example: accessing to \( z[a] \) is translated as
    \[
    \text{load}(id, [loc \mapsto \text{Addr}_z\.loc, \text{offs} \mapsto (\text{Addr}_z\.offs + (\text{load}(id, \text{Addr}_a) \times \text{Size}\_\text{of}\_\text{int})))}
    \]

- **Pointer arithmetic's and structure member**
  - The same kind computation is used to perform pointer arithmetics.
  - Similarly, accessing a structure member is achieved by shifting the base address of the structure with the constant accumulated size of all previous members.
  - Example: accessing to \text{student}.name is translated as
    \[
    \text{load}(id, [loc \mapsto \text{Addr}\_\text{student}.loc, \text{offs} \mapsto (\text{Addr}\_\text{student}.offs + \text{Offset}\_\text{student}.name)])
    \]
Each C function definition is translated into an operator with the process identifier $id$ as argument.

A C statement is translated into the conjunction of actions that are done simultaneously.

The function body is translated into the disjunction of the translation of each statement it contains.
Example

```
int process0(){
    int i = 1;
    x = x + i;
    return x;
}
```
Example

\[
\text{process0}(id) \triangleq \ \forall \ \text{head}(stack\_regs[id]).pc = \{\text{"process0"}, \text{"lbl_1"}\}
\land \text{store}(id, Addr_{process0_i}, [val \mapsto 1])
\land \text{stack\_regs'} = [stack\_regs \ \text{EXCEPT} \ ![id] = \langle pc \mapsto \{\text{"process0"}, \text{"lbl_2"}\},
\quad fp \mapsto \text{Head}(stack\_regs[id]).fp\rangle \circ \text{Tail}(stack\_regs[id])
\land \text{UNCHANGED} \ \langle ret \rangle
\]

... int process0()
{
  \ 1 \ int \ i = 1;
  \ 2 \ x = x + i;
  \ 3 \ return \ x;
}
Example

\[ \text{process0}(id) \triangleq \quad \forall \ \wedge \ \text{Head}(\text{stack\_regs}[id]).pc = \{\text{"process0"}, \text{"lbl\_1"}\} \]
\[ \wedge \ \text{store}(id, \text{Addr\_process0\_i}, [\text{val} \mapsto 1]) \]
\[ \wedge \ \text{stack\_regs}' = [\text{stack\_regs} \ \text{EXCEPT} \ ![id] = \{[pc \mapsto \{\text{"process0"}, \text{"lbl\_2"}\}, \]
\[ \quad fp \mapsto \ \text{Head}(\text{stack\_regs}[id]).fp]\} \circ \ \text{Tail}(\text{stack\_regs}[id])]] \]
\[ \wedge \ \text{UNCHANGED} \ \langle \text{ret} \rangle \]

... int process0(){
1  int i = 1;
2  x = x + i;
3  return x;
}
Example

\[
\text{process0}(id) \triangleq \begin{align*}
& \forall \ \wedge \text{Head}(\text{stack_regs}[id]).pc = \langle \text{"process0"}, \text{"lbl_1"} \rangle \\
& \wedge \text{store}(id, \text{Addr_process0}_i, [\text{val} \rightarrow 1]) \\
& \wedge \text{stack_regs}' = [\text{stack_regs EXCEPT !}[id] = \langle [\text{pc} \mapsto \langle \text{"process0"}, \text{"lbl_2"} \rangle, \\
& \quad \text{fp} \mapsto \text{Head}(\text{stack_regs}[id]).\text{fp} \rangle \circ \text{Tail}(\text{stack_regs}[id])]] \\
& \wedge \text{UNCHANGED} \langle \text{ret} \rangle \\
& \forall \ \wedge \text{Head}(\text{stack_regs}[id]).pc = \langle \text{"process0"}, \text{"lbl_2"} \rangle \\
& \wedge \text{store}(id, \text{Addr}_x, \text{plus}(\text{load}(id, \text{Addr}_x), \text{load}(id, \text{Addr_process0}_i)) \\
& \wedge \text{stack_regs}' = [\text{stack_regs EXCEPT !}[id] = \langle [\text{pc} \mapsto \langle \text{"process0"}, \text{"lbl_3"} \rangle, \\
& \quad \text{fp} \mapsto \text{Head}(\text{stack_regs}[id]).\text{fp} \rangle \circ \text{Tail}(\text{stack_regs}[id])]] \\
& \wedge \text{UNCHANGED} \langle \text{ret} \rangle \\
& \forall \ \wedge \text{Head}(\text{stack_regs}[id]).pc = \langle \text{"process0"}, \text{"lbl_3"} \rangle \\
& \wedge \text{stack_regs}' = [\text{stack_regs EXCEPT !}[id] = \text{Tail}(\text{stack_regs}[id])] \\
& \wedge \text{stack_data}' = [\text{stack_data EXCEPT !}[id] = \text{SubSeq}(\text{stack_data}[id], 1, \text{Head}(\text{stack_regs}[id]).\text{fp} - 1)] \\
& \wedge \text{ret}' = [\text{ret EXCEPT !}[id] = \text{load}(id, \text{Addr}_x)] \\
& \wedge \text{UNCHANGED} \langle \text{mem} \rangle
\end{align*}
\]

... int process0() {
  1 int i = 1;
  2 x = x + i;
  3 return x;
}
The translation of `goto/break/continue` statements consists in updating `stack_regs[id]` to the successor statement.

```plaintext
1  lbl0 : if (x < 10)
2    goto lbl1;
3  else goto lbl2;
4  lbl1: x ++;
5  goto lbl0;
6  lbl2: y = x;
7  ...
```
The translation of goto/break/continue statements consists in updating \texttt{stack_regs[id]} to the successor statement.

Example:

\begin{verbatim}
1  lb10 : if (x < 10)
2      goto lb11;
3  else goto lb12;
4  lb11: x ++;
5      goto lb10;
6  lb12: y = x;
7       ...
\end{verbatim}
The translation of goto/break/continue statements consists in updating \texttt{stack\_regs[id]} to the successor statement.

Example:

\begin{verbatim}
1  lbl0 : if (x < 10) 
2       goto lbl1;
3  elseif goto lbl2;
4  lbl1: x ++;
5  goto lbl0;
6  lbl2: y = x;
7  ...
\end{verbatim}
The translation of goto/break/continue statements consists in updating \texttt{stack_regs[id]} to the successor statement.

**Example:**

1. \texttt{lb10 : if (x < 10) goto lb11;}
2. \texttt{else goto lb12;}
3. \texttt{lb11: x ++; goto lb10;}
4. \texttt{lb12: y = x; goto lb10;}
5. ...
The translation of \texttt{goto/break/continue} statements consists in updating \texttt{stack_regs[id]} to the successor statement.

Example:

\begin{verbatim}
1  lb10 : if (x < 10)
2     goto lb11;
3  else goto lb12;
4  lb11: x ++;
5  goto lb10;
6  lb12: y = x;
7  ...
\end{verbatim}

\begin{verbatim}
\forall \land \mathit{Head}(\mathit{stack_regs[id]}).pc = \langle \text{"process0"}, \"lbl_1\" \rangle \\
\quad \land \mathit{IF} \ (\mathit{lt}(\mathit{load}(id, \mathit{Addr}_x, \langle \mathit{val} \mapsto 10 \rangle)) \neq \langle \mathit{val} \mapsto 0 \rangle) \\
\quad \quad \text{THEN } \mathit{stack_regs}' = [\mathit{stack_regs} \ except ![id] = \langle \mathit{pc} \mapsto \langle \text{"process0"}, \"lbl_2\" \rangle, \\
\quad \quad \quad \mathit{fp} \mapsto \mathit{Head}(\mathit{stack_regs[id]}.fp) \circ \mathit{Tail}(\mathit{stack_regs[id]}) \rangle] \\
\quad \quad \text{ELSE } \mathit{stack_regs}' = [\mathit{stack_regs} \ except ![id] = \langle \mathit{pc} \mapsto \langle \text{"process0"}, \"lbl_3\" \rangle, \\
\quad \quad \quad \mathit{fp} \mapsto \mathit{Head}(\mathit{stack_regs[id]}.fp) \circ \mathit{Tail}(\mathit{stack_regs[id]}) \rangle] \\
\quad \mathit{UNCHANGED} \ \langle \mathit{mem}, \mathit{stack_data}, \mathit{ret} \rangle \\
\forall \land \mathit{Head}(\mathit{stack_regs[id]}).pc = \langle \text{"process0"}, \"lbl_2\" \rangle \\
\quad \land \mathit{stack_regs}' = [\mathit{stack_regs} \ except ![id] = \langle \mathit{pc} \mapsto \langle \text{"process0"}, \"lbl_4\" \rangle, \\
\quad \quad \mathit{fp} \mapsto \mathit{Head}(\mathit{stack_regs[id]}.fp) \circ \mathit{Tail}(\mathit{stack_regs[id]}) \rangle] \\
\quad \mathit{UNCHANGED} \ \langle \mathit{mem}, \mathit{stack_data}, \mathit{ret} \rangle \\
\forall \land \mathit{Head}(\mathit{stack_regs[id]}).pc = \langle \text{"process0"}, \"lbl_3\" \rangle \\
\quad \land \mathit{stack_regs}' = [\mathit{stack_regs} \ except ![id] = \langle \mathit{pc} \mapsto \langle \text{"process0"}, \"lbl_6\" \rangle, \\
\quad \quad \mathit{fp} \mapsto \mathit{Head}(\mathit{stack_regs[id]}.fp) \circ \mathit{Tail}(\mathit{stack_regs[id]}) \rangle] \\
\quad \mathit{UNCHANGED} \ \langle \mathit{mem}, \mathit{stack_data}, \mathit{ret} \rangle \\
\forall \land \mathit{Head}(\mathit{stack_regs[id]}).pc = \langle \text{"process0"}, \"lbl_4\" \rangle \\
\quad \land \mathit{store}(id, \mathit{Addr}_x, \plus(\mathit{load}(id, \mathit{Addr}_x, \langle \mathit{val} \mapsto 1 \rangle)) \\
\quad \quad \text{AND } \mathit{stack_regs}' = [\mathit{stack_regs} \ except ![id] = \langle \mathit{pc} \mapsto \langle \text{"process0"}, \"lbl_5\" \rangle, \\
\quad \quad \quad \mathit{fp} \mapsto \mathit{Head}(\mathit{stack_regs[id]}.fp) \circ \mathit{Tail}(\mathit{stack_regs[id]}) \rangle] \\
\quad \mathit{UNCHANGED} \ \langle \mathit{ret} \rangle 
\end{verbatim}
The translation of goto/break/continue statements consists in updating \texttt{stack\_regs[id]} to the successor statement.

Example:

```plaintext
1 lb10 : if (x < 10)
2    goto lb11;
3 else goto lb12;
4 lb11: x ++;
5 goto lb10;
6 lb12: y = x;
7 ...
```
The translation of goto/break/continue statements consists in updating `stack_regs[id]` to the successor statement.

Example:

```plaintext
1  lbl0: if (x < 10)
2       goto lbl1;
3  else goto lbl2;
4  lbl1: x ++;
5  goto lbl0;
6  lbl2: y = x;
7  ...
```
All loops in C are normalized by CIL as a single while(1) looping construct that we translate like other jump statements.

C Example:

```c
while (x!=10) {
  x ++;
}
```

C code
All loops in C are normalized by CIL as a single while(1) looping construct that we translate like other jump statements.

C Example:

```c
while (1) {
  if (! (x != 10)) {
    goto while_0_break;
  }
  x ++;
}
while_0_break: ;
```

Normalized code
All loops in C are normalized by CIL as a single while(1) looping construct that we translate like other jump statements.

C Example:

```c
1 while (1) {
2   if (! (x != 10))
3     { goto while_0_break; }
4   x ++;
5 }
6 while_0_break: ;
```

Normalized code

\[
\begin{align*}
\forall \ & \text{Head}(\text{stack_regs}[id]).pc = \langle \text{"f1"}, \ "lb_1" \rangle \\
\land \ & \text{stack_regs}' = [\text{stack_regs} \ \text{EXCEPT} \ ![id]] = ([pc \mapsto \langle \text{"f1"}, \ "lb_2" \rangle, fp \\
\mapsto \text{Head}(\text{stack_regs}[id]).fp] \circ \text{Tail}(\text{stack_regs}[id])] \\
\land \ & \text{UNCHANGED} \langle \text{mem, stack_data, ret} \rangle
\end{align*}
\]
All loops in C are normalized by CIL as a single while(1) looping construct that we translate like other jump statements.

C Example:

```c
while (1) {
if (! (x != 10)) {
    goto while_0_break;
}
    x ++;
while_0_break: ;
```
All loops in C are normalized by CIL as a single while(1) looping construct that we translate like other jump statements.

C Example:

```c
while (1) {
    if (! (x != 10)) {
        goto while_0_break;
    }
    x ++;
    while_0_break:
}
```

Normalized code:

```
∧ Head(stack_regs[id]).pc = ⟨“f1”, “lbl_1”⟩
∧ stack_regs' = [stack_regs EXCEPT ![id] = ⟨pc ↦ ⟨“f1”, “lbl_2”⟩, fp
                          ↦ Head(stack_regs[id]).fp⟩ ○ Tail(stack_regs[id])]
∧ UNCHANGED ⟨mem, stack_data, ret⟩
∧ Head(stack_regs[id]).pc = ⟨“f1”, “lbl_2”⟩
∧ IF (ne((load(id, [loc ↦ Addr_x.loc, offs ↦ Addr_x.offs])), ([val ↦ 10]) ≠ [val ↦ 0])
  THEN stack_regs' = [stack_regs EXCEPT ![id] = ⟨pc ↦ ⟨“f1”, “lbl_4”⟩,
                              fp ↦ Head(stack_regs[id]).fp⟩ ○ Tail(stack_regs[id])]
  ELSE stack_regs' = [stack_regs EXCEPT ![id] = ⟨pc ↦ ⟨“f1”, “lbl_3”⟩,
                              fp ↦ Head(stack_regs[id]).fp⟩ ○ Tail(stack_regs[id])]
∧ UNCHANGED ⟨mem, stack_data, ret⟩
∧ Head(stack_regs[id]).pc = ⟨“f1”, “lbl_3”⟩
∧ stack_regs' = [stack_regs EXCEPT ![id] = ⟨pc ↦ ⟨“f1”, “lbl_6”⟩,
                           fp ↦ Head(stack_regs[id]).fp⟩ ○ Tail(stack_regs[id])]
∧ UNCHANGED ⟨mem, stack_data, ret⟩
```
All loops in C are normalized by CIL as a single while(1) looping construct that we translate like other jump statements.

C Example:

```
while (1) {
  if (! (x != 10)) {
    goto while_0_break;
  }
  x ++;
}
while_0_break: ;
```

Normalized code

```
∀ Head(stack_regs[id]).pc = 〈“f1”, “lbl_1”〉
∧ stack_regs' = stack_regs EXCEPT ![id] = 〈[pc ↦ (“f1”, “lbl_2”), fp
→ Head(stack_regs[id]).fp] ∘ Tail(stack_regs[id])〉
∧ UNCHANGED 〈mem, stack_data, ret〉
∀ Head(stack_regs[id]).pc = 〈“f1”, “lbl_2”〉
∧ IF (ne((load(id, [loc ↦ Addr_x.loc, offs ↦ Addr_x.offsets])), ([val ↦ 10])) ≠ [val ↦ 0])
THEN stack_regs' = stack_regs EXCEPT ![id] = 〈[pc ↦ (“f1”, “lbl_4”),
fp ↦ Head(stack_regs[id]).fp] ∘ Tail(stack_regs[id])〉
ELSE stack_regs' = stack_regs EXCEPT ![id] = 〈[pc ↦ (“f1”, “lbl_3”),
fp ↦ Head(stack_regs[id]).fp] ∘ Tail(stack_regs[id])〉
∧ UNCHANGED 〈mem, stack_data, ret〉
∀ Head(stack_regs[id]).pc = 〈“f1”, “lbl_3”〉
∧ stack_regs' = stack_regs EXCEPT ![id] = 〈[pc ↦ (“f1”, “lbl_6”),
fp ↦ Head(stack_regs[id]).fp] ∘ Tail(stack_regs[id])〉
∧ UNCHANGED 〈mem, stack_data, ret〉
∀ Head(stack_regs[id]).pc = 〈“f1”, “lbl_4”〉
∧ store(id, Addr_x, plus(load(id, Addr_x), [val ↦ 1]))
∧ stack_regs' = stack_regs EXCEPT ![id] = 〈[pc ↦ (“f1”, “lbl_1”),
fp ↦ Head(stack_regs[id]).fp] ∘ Tail(stack_regs[id])〉
∧ UNCHANGED 〈ret〉
```
All loops in C are normalized by CIL as a single while(1) looping construct that we translate like other jump statements.

C Example:

```c
while (1) {
  if (! (x != 10)) {
    goto while_0_break;
  }
  x ++;
}
while_0_break: ;
```

Normalized code

```plaintext
\( \land \text{Head}(\text{stack_regs[id]}).pc = \{"f1", "lbl_1"}\)\n\( \land \text{stack_regs}' = [\text{stack_regs EXCEPT } ![id] = \{[pc \mapsto \{"f1", "lbl_2"}, fp \mapsto \text{Head}(\text{stack_regs[id]}).fp\} \circ \text{Tail}(\text{stack_regs[id]})]\)\n\( \land \text{UNCHANGED} \{\text{mem, stack_data, ret}\}\)
\( \land \text{Head}(\text{stack_regs[id]}).pc = \{"f1", "lbl_2"}\)\n\( \land \text{if } (\text{ne}(\text{load}(id, [loc \mapsto \text{Addr}_x.loc, offs \mapsto \text{Addr}_x.offs]), ([\text{val} \mapsto 10]) \neq [\text{val} \mapsto 0])\)\n\( \land \text{THEN } \text{stack_regs}' = [\text{stack_regs EXCEPT } ![id] = \{[pc \mapsto \{"f1", "lbl_4"}, fp \mapsto \text{Head}(\text{stack_regs[id]}).fp\} \circ \text{Tail}(\text{stack_regs[id]})]\)\n\( \land \text{ELSE } \text{stack_regs}' = [\text{stack_regs EXCEPT } ![id] = \{[pc \mapsto \{"f1", "lbl_3"}, fp \mapsto \text{Head}(\text{stack_regs[id]}).fp\} \circ \text{Tail}(\text{stack_regs[id]})]\)\n\( \land \text{UNCHANGED} \{\text{mem, stack_data, ret}\}\)
\( \land \text{Head}(\text{stack_regs[id]}).pc = \{"f1", "lbl_3"}\)\n\( \land \text{stack_regs}' = [\text{stack_regs EXCEPT } ![id] = \{[pc \mapsto \{"f1", "lbl_6"}, fp \mapsto \text{Head}(\text{stack_regs[id]}).fp\} \circ \text{Tail}(\text{stack_regs[id]})]\)\n\( \land \text{UNCHANGED} \{\text{mem, stack_data, ret}\}\)
\( \land \text{Head}(\text{stack_regs[id]}).pc = \{"f1", "lbl_4"}\)\n\( \land \text{store}(id, \text{Addr}_x, \text{plus}(\text{load}(id, \text{Addr}_x), [\text{val} \mapsto 1]))\)\n\( \land \text{stack_regs}' = [\text{stack_regs EXCEPT } ![id] = \{[pc \mapsto \{"f1", "lbl_1"}, fp \mapsto \text{Head}(\text{stack_regs[id]}).fp\} \circ \text{Tail}(\text{stack_regs[id]})]\)\n\( \land \text{UNCHANGED} \{\text{ret}\}\)
\( \land \text{Head}(\text{stack_regs[id]}).pc = \{"f1", "lbl_6"}\)\n\( \land \text{stack_regs}' = [\text{stack_regs EXCEPT } ![id] = \{[pc \mapsto \{"f1", "lbl_7"}, fp \mapsto \text{Head}(\text{stack_regs[id]}).fp\} \circ \text{Tail}(\text{stack_regs[id]})]\)\n\( \land \text{UNCHANGED} \{\text{mem, stack_data, ret}\}\)
```
A function call is translated in two actions:
- The stack frame is pushed onto the `stack_data[id]` which obeys the LIFO order.
- The `stack_regs[id]` is updated by changing its head to a record whose `pc` field points to the action done once the call has finished.
- On top of `stack_regs[id]` is pushed a record with `pc` pointing to the first statement of the called function, and `fp` to the new stack frame.

```plaintext
int max(int i, int j)
{
  if (i => j)
    return i;
  else return j;
}

void f1()
{
  x = ... 
  y = ...
  int m = max(x, y);
  ...
}
```
A function call is translated in two actions:

- The stack frame is pushed onto the `stack_data[id]` which obeys the LIFO order.
- The `stack_regs[id]` is updated by changing its head to a record whose `pc` field points to the action done once the call has finished.
- On top of `stack_regs[id]` is pushed a record with `pc` pointing to the first statement of the called function, and `fp` to the new stack frame.

```plaintext
int max(int i, int j) {
    if (i => j) return i;
    else return j;
}

void f1() {
    x = ...
    y = ...
    int m = max(x, y);
    ...
}
```
A function call is translated in two actions:

- The stack frame is pushed onto the stack_data[id] which obeys the LIFO order.
- The stack_regs[id] is updated by changing its head to a record whose pc field points to the action done once the call has finished.
- On top of stack_regs[id] is pushed a record with pc pointing to the first statement of the called function, and fp to the new stack frame.
- The second action copies the return value ret[id] in the considered variable.

```
int max(int i, int j)
{
  if (i => j)
    return i;
  else return j;
}

void f1()
{
  x = ...;
  y = ...;
  int m = max(x, y);
  ...}
```
Once the function returns:

- The top of the `stack_regs[id]` is popped,
- Its stack frame is removed from stack data[id] (using the `SubSeq` operator).
- The returned value is stored on `ret[id]`.

Example:

```c
... int process0(){
    int i = 1;
    x = x + i;
    return x;
}
```

\[ \land \ Head(stack_regs[id]).pc = \{\text{"process0"}, \text{"lbl_3"}\} \]
\[ \land stack_regs' = [stack_regs \ EXCEPT ![id] = Tail(stack_regs[id])] \]
\[ \land stack_data' = [stack_data \ EXCEPT ![id] = SubSeq(stack_data[id], 1, Head(stack_regs[id]).fp - 1)] \]
\[ \land ret' = [ret \ EXCEPT ![id] = load(id, Addr_x)] \]
\[ \land \text{UNCHANGED} \langle \text{mem} \rangle \]
Generating the specification

- **Init** predicate that initializes all variables of the system.
  - The number of process and the entry point (initial function) of each one are specified by user. This will initialize the `stack_regs` variable.
  - The `mem` variable is initialized according to the initializers of global variables.
  - The `stack_data` is initially empty and the `ret` variable contains `Undef` value, for all processes.

- The predicate `process(id)`, that defines the next-state action of the process `id`.
  - It asserts that one of the function is being executed while `stack_reg[id]` is not empty.

\[
\text{process}(id) \triangleq \wedge \text{stack_regs}[id] \neq \langle \rangle \\
\wedge (\forall \text{max}(id) \lor \text{process0}(id) \lor \text{process1}(id))
\]

- The tuple of all variables

\[
\text{vars} \triangleq \langle \text{mem}, \text{stack_regs}, \text{stack_data}, \text{ret} \rangle
\]
The next-state action $\text{Next}$ of all processes

One of the process that has not finished is nondeterministically chosen to execute one step until $\text{stack\_regs}$ becomes empty.

\[
\text{Next} \triangleq \forall \exists \ id \in \text{Procs} : \text{process}(id) \land \forall \ id \in \text{Procs} : (\text{stack\_regs}[id] = \langle \rangle) \land (\text{UNCHANGED \ vars})
\]

The complete specification

\[
\text{Spec} \triangleq \text{Init} \land \square[\text{Next}]_{\text{vars}}
\]
Examples of properties

```java
1. int x = 0;
2. int lock_var = 0; // lock global variable
3. void process0(int i){
4.   acquire_mutex();
5.   x++;
6.   x = x + i;
7.   release_mutex();
8.   ...}
9. }
10. void process1(int j){
11.   acquire_mutex();
12.   j = x;
13.   x = x + i;
14.   release_mutex();
15.   ...}
```
Examples of properties

Mutual exclusion

∀sc1 ∈ {("process0", "lbl_6"), ("process0", "lbl_7")}: ∀sc2 ∈ {("process1", "lbl_13"), ("process1", "lbl_14")}: ((Head(stack_regs["process0"]).pc = sc1) ⇒ (Head(stack_regs["process1"]).pc ≠ sc2))
Examples of properties

```
1. int x = 0;
2. int lock_var = 0; // lock global variable
3. void process0(int i){
   4.   acquire_mutex();
   5.   x++;
   6.   x = x + i;
   7.   release_mutex();
   8.   ...
3. }
10. void process1(int j){
   11.   acquire_mutex();
   12.   j = x;
   13.   x = x + i;
   14.   release_mutex();
16.   ...
```

- **Mutual exclusion**

\[
\forall sc1 \in \{\text{“process0”, “Ibble.6”}, \text{“process0”, “Ibble.7”}\}:
\forall sc2 \in \{\text{“process1”, “Ibble.13”}, \text{“process1”, “Ibble.14”}\}:
(Head(stack_regs[“process0”]).pc = sc1) \Rightarrow
(Head(stack_regs[“process1”]).pc \neq sc2)
\]

- **Termination**

\[
\Diamond (\forall id \in \{\text{“process0”, “process1”}\} : Head(stack_regs[id]).pc = ())
\]

- Considering fairness assumptions in the specification
Conclusion

- C2TLA+: A translator from C to TLA+ specification that can be checked by TLC.
- The translation is based on a set of translation rules.

Future works

- Handle missing features.
- Catch all C runtime-errors.
- Using TLAPS to prove that a (translated) specification implements an abstract one.
C2TLA+: A translator from C to TLA+

Amira METHNI (CEA/CNAM)
Matthieu LEMERRE (CEA) & Belgacem BEN HEDIA (CEA)
Serge HADDAD (ENS Cachan) & Kamel BARKAOUI (CNAM)
Thank you

Questions ??
C2TLA+: A translator from C to TLA+

Amira METHNI (CEA/CNAM)
Matthieu LEMERRE (CEA) & Belgacem BEN HEDIA (CEA)
Serge HADDAD (ENS Cachan) & Kamel BARKAOUI (CNAM)

June 3, 2014