1 High-level rules of translation

1.1 Translator input

The input is a TLA+ Spec that has the following properties.

1.1.1 TypeOK

The TLA+ Spec is supposed to contain an invariant *TypeOK*. This predicate is required to obtain the types of variables. In (1), the objects $S\_1 \ldots S\_n$ are sets of integers boolean or arrays (of integers or booleans). Notice that the formula $xi = value\_i$ is equivalent to $xi \in Set\_i$ with Set_i is a singleton.

```
VARIABLES x1,.., xn

TypeOK ==
          /\ x1 \in Set_1
          ....
          ....
          /\ xn \in Set_n
```

Listing 1: The explicit form of the predicate *TypeOK*.

1.1.2 Proc

This constant is used to declare the processes identifiers. The user is supposed to define it as a part of the input.

1.2 Initial state

Suppose that the spec has the variables $x1, \ldots, xn$ of types George: still not the complete list of types. Assume that the initial state is of the form:

```
Init == /\ x1= Value_1
        /\ x2= Value_2
        .....
        .....
        /\  xn= Value_n
```

We are going to describe in details how to translate the initial state depending on the "shape" of $Value_i$: Let h be the maximal dimension of the tuple variables, i.e., $h = \max\{dim(xi) \mid 1 \leqslant i \leqslant n\}$[1] The translation of *Init* is parametrized by h variables, that is, the Cubicle intial state takes the form

[1] By dimension we mean the minimum number of indexes needed to parametrize xi.

```
init (z1 ... zh) { Trans(x1= Value_1)
                   &&  Trans(x2= Value_2)
                    ...
                   && Trans(xn= Value_n)}
```

Now we determine (when possible) more explicitly how to compute *Trans(x2= Value_2)*. We consider the following cases:

(a) $Value\_i$ is an integer or boolean value, then the translation is straightforward. For the case where $Value\_i$ is a string, an abstract type is already defined with $Value\_i$ is one of its values $\star$.

George: will refer to the related detailed part once it is ready

(b) $Value\_i$ = [$x \in C$ |-> $Target_i$], where C is a constant in the Spec and $Target_i$ is a constant function (not necessarily declared as a constant) with the domain C. Then, $xi = Value\_i$ is translated to

$xi[Xi] = Target_i[X_i]$, where $X_i = \{z1 \ldots zi\}$ and $i = dim(C)$. A condition to check is $dim(C) = dim(xi)$. The type of $Target_i[X_i]$ is integer, boolean or string $\star$. For the case where $Target_i[X_i]$ is a string, an abstract type is already defined with $Target_i[X_i]$ is one of its values $\star$.

George: still investigating about the other data types

George: will refer to the related detailed part once it is ready

(c) For a constant C, *case_1, . . . , case_k* are state predicates, and *Value_i1, . . . , Value_ik* are state functions, if xi is initialized as follows:

```
xi = [self \in C |-> CASE case_1 -> Value_i1
                 [] case_2 -> Value_i2
                      ....
                 [] case_k -> Value_ik
```

then the translation is of the form:

```
(x_1[X1] =  Trans(Value_i1)  &&  Trans(case_1) )
|| (x_2[X2] =  Trans(Value_i2)  &&  Trans(case_2))
   ...
   ...
|| (x_k[Xk] =  Trans(Value_ik)  &&  Trans(case_k))
```

See $\star$ for the translation of state predicates and state functions.

George: refer to the corresponding part once it is ready

(d) As a special case xi \in $\{e1, \ldots, ek\}$, where ei is of type integer, boolean or string. $\star$ Then, the translation is

George: still investigating about the other data types

```
   x_i =  e1
   || x_i =e2
   ...
   ...
   || x_i=ek
```

1.3 Next state

Assume that the Next predicate is of the form:

```
Next == \/  N1
        \/  N2
            ...
            ...
        \/  N_k
```

where Ni is one of the following:

(a) $Ni = P_p/\backslash(xi' = P_n)$, with P_n and $P_p$are state function and state predicate respectively $\star$ Then, the translation is of the form:

George: reminder: Think about the case where P_n is a next-state relation

```
transition Ni()
requires {  Trans(P_p) }
{ xi:= Trans(P_n);}
```

$Trans(P_p)$ and P_n can be computed as in $\star$.

George: will refer to the related detailed part once it is ready

(b) $Ni = \backslash E\ \ z \backslash in\ Proc : P_p(z)/\backslash(xi' = P_n(z))$, with P_n and P_p are state function and state predicate respectively $\star$ Then, the translation (with the abuse of notation for z) is of the form: [2]

George: reminder to me: Think about the case where P_n is a next-state relation

```
transition Ni(z)
requires {  Trans(P_p(z)) }
{ xi:= Trans(P_n(z));}
```

(c) A generalization of (b) is when z is a subset of $Proc$. Then, the translation is analogous to the earlier case.

1.4 Predicate & Quantified logic

We discuss here the translation of sate functions, i.e., expressions containing variables and constants that are not next-state relations (no primed variables).

We consider the following cases:

1.4.1 Simple date types

For a variable x of a simple data type (integer, boolean or string)

(a) The simple (sub-)expression $x = value$, where x is a variable and $value$ is an integer or a boolean constant. In this case, the translation is simple and straightforward.

[2]Clearly, the sub-formula $(xi' = P_n(z))$ in this case can be generalized to the multi-variable one.

(b) For the (sub-)expression $x = value$, where $value$ is a string, a Cubicle abstract type x_values is defined that encodes all values that x takes in the whole TLA-Spec.

(i) If an invariant of the TLA-spec implies that x takes values on a finite set of strings, that is, $x \in StringSet$, with $StringSet = \{S1, \ldots, Sk\}$, then x_values is defined as:

```
type x_type = S1 | S2 | ... | Sk
var x_var : x_type
```

After that, every (sub-)expression of the form $x = Si$ is translated to

$$x\_var = Si;$$

$\star$

George: I agree that it is a complicated way, but I am unable to find a more simple one.

(ii) Otherwise, we scan the whole TLA-spec looking for (sub)expressions of the form $x = value$ or $x' = value$ computing a set $StringSet$ that is dealt with the same way of the previous case.

1.4.2 Complex date types

We restrict our selves to the following cases:

(a) For the (sub-)expression $x = Value$, where $Value$ of the form:

$$[c1 \backslash in\ C1 ... ck \backslash in\ Ck\ |->Value_{c1,\ldots,ck}],$$

where $Value_{c1,\ldots,ck}$ is an integer, a boolean or a string value. $x = Value$ is translated to:

$$x(z1...zk) = Translation(Value_{c1,\ldots,ck})$$

where $Translation(Value_{c1,\ldots,ck})$ is computed as in Sec 1.4.1.

(b) For the (sub-)expressions $x = Value$, $x' = Value$ where $Value$ of the form:

$$[Value_{old}\ EXCEPT\ !c1 = e1, ..., !.cn = ek\ \ |->Value_{c1,\ldots,ck}],$$

where $Value_{c1,\ldots,ck}$ is an integer, a boolean or a string value and $Value_{old}$ is a constant record of the same type of the one defined in (a), then the expression $x = Value$ is translated to:

$$x(z1...zk) = case \mid z1 = e1\ ...\ zk = ek\ \ : Value_{c1,\ldots,ck}$$

$$|\_ : Value_{old}[z1...zk]$$

(c) For the case $x =<< c1, ...cm >>$ we do as in (a) considering $x = [i \in 1..m|->x[i]]$. Notice the multi-dimensional case can be handled analogously.

1.4.3 Locating the translation of state predicates

Depending on the position of the TLA-expressions, we its the translation:

(a) (sub-)expressions appearing in the TLA++ initial state are translated to predicates in the initial Cubicle one.

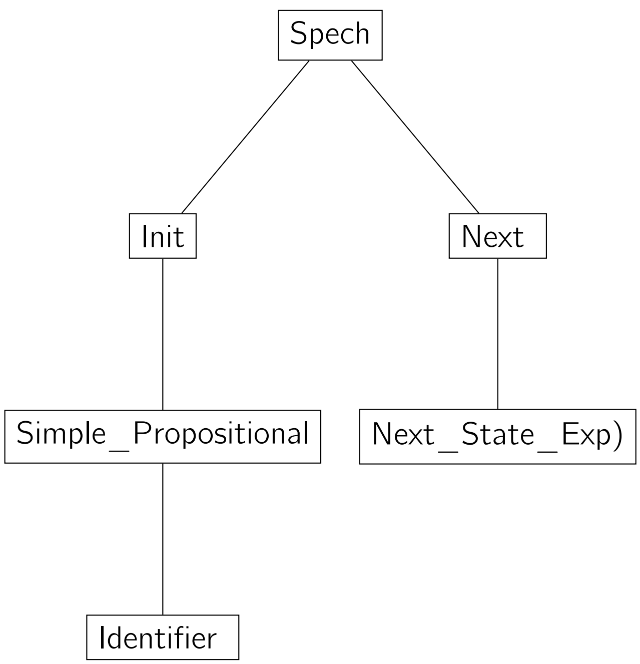
(b) The translation of (sub-)expressions of invariant negations are located in the unsafe predicate in the Cubicle Spec.

(c) The (sub-)expressions in the next state are translated to the *requesters* part of a Cubicle *transition*.

2 Grammar of the fragment

In this section, we define the grammar of the fragment we are interested in:

George: Illustration of the grammar of the file $Trans\_Input\_Gram.tla$. Not ready yet



3 Translating algorithm

Input : A TLA++ Spec based on the grammar of Section 2 with *TypeOk* (Section 1.1.1 invariant and constant *Proc* (Section 1.1.2) as in Section.

Output: A Cubicle Spec that is equivalent to the one in the input.

`prepa()` **for** *all variables* x_i **do**

- Compute all $Type(x_i)$ from the TypeOk Invariant
- **if** $Type(x_i) = String$ **then**
  - Define an abstract Cubicle-type that involves all possible values of x_i
  - George: Reminder to me: refer to the corr. part.

Translating Init

- // Recall that Init == $\bigwedge_{i=1}^{n} x_i = Value_i$
- **for** $i = 1$ **to** n **do**
  - **if** $Type(Value) \in \{Integer, Boolean\}$ **then**
    - $Translation(x_i = Value_i) := x_i = Value_i$
  - **else if** $Value_i = "Text_i"$ **then**
    - $Translation(x_i = Value_i) := x_i = Text_i$
  - // In the following, $Proc\_SubSet$ is a subset of $Proc$
  - **else if** $Value_i = [z \setminus in\ Proc\_SubSet\ |-> Value_{i,z}]$ **then**
    - // with $Type(Value_{i,z}) \in \{String, Boolean, String\}$
    - $Translation(x_i = Value_i) := x_i(z) = Value_{i,z}$
    - George: Will consider multi-dim functions later
- The translation of Init is:

```
init (z)   { Translation(x1= Value_1)
             &&  Translation(x2= Value_2)
               ...
             && Translation(xn= Value_n)}
```
;

Translating Next

- // Recall that Next == $\bigvee_{i=1}^{k} (Predicate_i \bigwedge (x_i' = Value_i))$
- K' is an inclusion-wise maximal subset of $\{1, \ldots, k\}$ with $Predicate_i, Predicate_j$ are not equivalent for all $1 \leqslant i \neq j \leqslant k'$
- **foreach** $i \in K'$ **do**
  - Define a Cubicle-transition T_i as follows:
  - **Defining Transition** T_i
    - Compute $Translation(Predicate_i)$ as in Section 1.4 and place it in T_i "requires"-part
    - **if** $Type(Value_i) \in \{Integer, Boolean\}$ **then**
      - $Translation(x_i' = Value_i) := (x_i := Value_i)$
    - **else if** $Value_i = "Text_i"$ **then**
      - $Translation(x_i = Value_i) := (x_i =: Text_i)$
    - **else if** $[Value_{old}\ EXCEPT\ ![c] = \vec{e}\ ]$ **then**

```
x(z) = case |  z = c : e
            |_ : Value_{old}[z]
```

Algorithm 1 TLA++ - Cubicle Translator

(13) The translation of Next is:

(14)
```
transition Ti (z)
requires {      (* with {j1, ... , jh} = {1 <=j<= k,  Predicate_j =Predicate_i } a *)
            Translation(Predicate_i)    }
          {
          Translation(xj1=Valuej1);
            ...
            ...
          Translation(xj1=Valuejh);
          }
```
;

Translating invariant

(15) Translate the negation of each invariant $Invar_1, \ldots, Invar_s$ as in Section 1.4 ;

(16) Define unsafe (z) as:

```
unsafe (z) = Translating(~ Invar_1)
             || Translating(~ Invar_2)
                ...
                ...
             ||  Translating(~ Invar_s)
```
;

return Cubicle-spec with the computed *unsafe (z), Init (z)* and $T_1(z), \ldots, T_{|K'|}(z)$.