1 High-level rules of translation

This section is dedicated to describe the rules of the translator we define. Based on these rules.

1.1 Translator input

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

1.1.1 Proc

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

1.1.2 TypeOK

As mentioned before, the TLA+ Spec is supposed to contain an invariant *TypeOK*. This predicate is required to obtain the types of variables. In Listing (1), the objects $Set\_1 \dots Set\_n$ are sets of integers boolean or function sets. For the last case, Set_i is of the form $[Proc\_Subset \text{ –> } Target_i]$ where $Proc\_Subset$ is a subset of $Proc$ and $Target_i$ is a set of integers, boolean or strings. Notice that the formula $xi = value\_i$ is equivalent to $xi \in Set\_i$ with Set_i is a singleton. Hence, we allow $xi = value\_i$ syntactically [1].

```
VARIABLES x1,.., xn

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

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

1.1.3 Structure

We are going to assume that the TLA+ Spec has a simple structure. In particular, no labeled predicates are used as sub-expressions of (resp. next-) state relations or invariants. If any, we simply expand all expressions to get the following:

```
CONSTANTS c1, .. , cm


VARIABLES x1,.., xn

```

[1] Note that for the case of functions, we have the form $xi = [Obj \ \backslash in \ Proc\_Subset \ | \text{–>} \ \text{Expression}(Obj)]$

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

Invariants == (* Defining the invariants of the Spec *)

(* I_i is a state predicate that determines the initial state of x_i *)

Init == /\ I_1
        /\ I_2
        .....
        .....
        /\ I_n

(* N_i is a next-state relation (more details are below *)

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

Spec == Init \/ []Next_<<x1, .. xn>>
```

We are going to give a more precise form later on.

1.2 Predicate & Quantified logic

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

We consider the following cases:

1.2.1 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.

(b) For the (sub-)expression $x = value$, where $value$ is a string, a Cubicle abstract type is defined that encodes all values of string-type variables. To compute the values, we recognize two cases:

   (i) If the TypeOk invariant states that x takes values on a finite set of strings, that is, $x \in StringSet_x$, then we are done with this case.

(ii) Otherwise, we scan the whole TLA-spec looking for (sub)expressions of the form $x = value$ or $x' = value$ computing a set $StringSet_x$.

After computing StringSet of all string-type variables, we define String_values $= \{S1, \ldots, Sk\}$ that is the union of all $StringSet_x$. In the Cubicle spec, we define an abstract type $String\_type$ and x_i for each string-type variable as follows:

```
String_type = S1 | S2 | ... | Sk
var x : String_type
```

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

$$x\text{= } Si;$$

1.2.2 Complex date types

We restrict our selves to the following cases:

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

$$[c1...ck \setminus in\ Proc\_Subset\ \ |\text{-->}\ Value_{c1,\ldots,ck}],$$

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 1.2.1.

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

$$[Value_{old}\ EXCEPT\ ![c1] = e1, ..., ![ck] = ek\ ],$$

where $e1, \ldots, ek$ are integers, booleans or strings values and $Value_{old}$ is a constant function of the same type of the one defined in (a), then the expression $x = Value$ is translated to:

$$\begin{aligned}
x(z) = case \\
&| z = c1 : e1 \\
&| z = c2 : e2 \\
&\ldots\ldots \\
&| z = ck : ek \\
&|_ : Value_{old};
\end{aligned}$$

1.2.3 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*.

1.3 Initial state

Suppose that the spec has the variables $x1, \ldots, xn$ of types determined by the invariant TypeOK as described in Sec 1.1.2. We consider the following form of the initial state:

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

George: The blue parts of the rest concern the multi-dimensional case. I keep them, however, I ignore them in the algorithm and the grammar for the moment. 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., $h = \max\{dim(xi) \mid 1 \leqslant i \leqslant n\}$[2] The translation of *Init* is parametrized by h variables, that is, the Cubicle intial state takes the form

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

Now we determine (when possible) more explicitly how to compute *Trans(xi= Value_i)*. 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 (Section 1.2).

(b) $Value\_i$ = [*Obj* \in $Proc\_Subset$ |–> $Target_i$], where $Target_i$ is a constant function (not necessarily declared as a constant[3]) with the domain $Proc\_Subset$. Then, $xi = Value\_i$ is translated to

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

[3] For example, $Value\_i$ = [*Obj* \in $Proc\_Subset$ |–> k] with k is an integer.

$xi[Xi] = Target_i[X_i]$, where $X_i = \{z1 \dots zi\}$ and $i = dim(Proc\_Subset)$. A condition to check is $dim(Proc\_Subset) = dim(xi)$. The type of $Target_i[X_i]$ is integer, boolean or string. Again, 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 (Sec 1.2).

(c) For a constant $Proc\_Subset$, *case_ 1, ..., case_ k* are state predicates, and *Value_ i1, ..., Value_ ik* are state functions of integer, boolean or string type, if xi is initialized as follows:

```
xi = [self \in Proc\_Subset |-> 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 Sec 1.2 for the translation and the restrictions on the state predicates *case_ 1, ..., case_ k* and state functions.

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

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

1.4 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. Then, the translation is of the form:

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

$Trans(P_p)$ and P_n can be computed as in Sec 1.2

(b) $Ni = \backslash E \;\; z \, \backslash in \; Proc\_Subset : P_p(z) / \backslash (xi' = P_n(z))$, with P_n and P_p are state function and state predicate respectively. Then, the translation (with the abuse of notation for z) is of the form: [4]

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

(c) George: The idea of Case(c) is a remark that you mentioned on a meeting. I hope that I understood it collectedly. I still do not considered it in the Algo, until you confirm it. Suppose we have that:

```
Ni== \E z \in Proc : (\A y \in Proc : x[y]'= Value_y)
```

where $Value_y$ is an integer, boolean or string. Then the translation of Ni is computed as in the case:

```
Ni== x'=[y \in  |-> Value_y ]
```

(d) A generalization of (b) is the multivariable case $z1, \dots, zk$ can be analogously done.

2 Grammar of the fragment

Since the illustration of codes below is still too bad, I suggest to read the grammar details in "$TlaplusToCubicle/grammer/Trans\_Input\_Gram.tla$" until I find a way to improve it as a Latex environment.

```
------------------------------ MODULE FragGram ------------------------------
( * We use  BNFGrammars https://github.com/tlaplus/Examples/blob/master/specifications/SpecifyingSyste



Spec == Init
        & tok("\/")
        & tok("[][")
        & Next
        & tok("]_")
        & VARIABLES

```

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

```
VARIABLES == tok("<<")
             & ( Identifier
             & (tok(",") & Identifier)^* )
             & tok(">>")


(* ############################################################################################# *)
(* ########### Defining the Init part ########################################################## *)
(* ############################################################################################# *)

Init == (tok('/\') & Simple_Propositional_Exp)^+
Simple_Propositional_Exp ==  (Identifier & tok("=") & VALUE)
                             |  (Identifier & tok("\in") & Finite_Set)

Finite_Set == tok('{')
                & Value
                & (tok(",") & Value)^*
                &  tok('}')
              | Numeral^+ & tok("..") & Numeral^+
              | Numeral^+ & tok("..") &  Identifier
              | Identifier & tok("..") &  Numeral^+
              | Identifier & tok("..") &  Identifier


VALUE == Identifier (* I have a constant in mind *)
         | Numeral^+
         | Numeral^+ & tok(".") & Numeral^*
         | STRING
         | Boolean
         | Function
         | Identifier  & tok("[") & (TERM) & tok("]")
         | Identifier  & tok("(") & (TERM) & tok(")")


Function == tok('[')
                & Identifier
                & tok(' \in ')
                & Identifier  (* I have a subset of Proc in mind*)
                & tok('|->')
                & TERM
                & tok(']')


(* In the following, Tok(P) means that the set of all terms of the form P  *)
STRING == Tok (tok(" " ") & NameChar^* & tok(" " ") )  \ ReservedWords (* The same ReservedWords defin

Boolean == tok("True") | tok("False")
```

```
Identifier == Name \ ReservedWords
Name == Tok((NameChar^* & Letter & NameChar^*))
NameChar  == Letter \cup Numeral \cup {"_"}
Letter == OneOf("abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ")
Numeral   == OneOf("0123456789")


(* ############################################################################################# *)
(* ########### Defining the Next part ########################################################## *)
(* ############################################################################################# *)

Next ==  (tok('\/') & Next_State_Exp)^+


Next_State_Exp ==    (tok('/\') & Predicate)^*
                     & (tok('/\') & Primed_Exp)^+

(* ############################################### *)
(* ########### Defining Predicate ################ *)
(* ############################################### *)

 Predicate ==  Propositional_Exp
               |  tok("(")
                  & (tok('\E') | tok('\A'))
                  & Identifier (* For the moment, let us stay in the one-variable case *)
                  & tok("\in")
                  & Identifier  (* I have a subset of Proc in mind *)
                  & tok(' : ')
                  & Predicate
                  & tok(")")

Propositional_Exp ==  Simple_Propositional_Exp
                      | tok("(") & TERM & OPERATOR & TERM & tok(")")
                      | tok('~') & tok("(")   & Propositional_Exp & tok(")")
                      | tok("(")
                        & Propositional_Exp
                        & Logical_Junctions
                        & Propositional_Exp
                        & tok(")")


OPERATOR == tok("=")  |  tok("#") | tok("~=") | tok("<")  | tok("<=")


Logical_Junctions == tok("/\") | tok("\/")


TERM == VALUE | Open_Prpos

```

```
Open_Prpos == Identifier
             | Identifier  & tok("[") & TERM & tok("]")
             | Identifier  & tok("(") & TERM & tok(")")
             (* generalize it more? *)

(* ############################################### *)
(* ########### Defining Primed_Exp ################ *)
(* ############################################### *)


Primed_Exp ==   Identifier
                & tok(' ' ')
                & tok("=")
                & (TERM | Function_Except)

Function_Except == tok("[")
                   &  Identifier
                   & tok("EXCEPT")
                   & ( tok('![') &  Identifier & tok(']=') & TERM
                     & (tok('![') &  Identifier & tok(']=') & TERM )^+ )
                   & tok("]")
```

3 Translating algorithm

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

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

Data-Type Preparation

Simply structure as mentioned in Sec 1.1.3

String_values := {}

for *all variables* x_i **do**

  Compute all $Type(x_i)$ from Invariant TypeOk

  if $Type(x_i) = String$ **then**

    Find all possible values of x_i and add them to String_values. This can be obtained from TypeOk invariant or scanning all possible values of x_i as described in Sec 1.2.1.

Translating Init

// Recall that Init == $\bigwedge_{i=1}^{n} (x_i = Value_i$ or $x_i \in \text{Finite\_set}_i)$

for $i = 1$ **to** n **do**

  if $Type(Value) \in \mathbb{Z} \cup \{True, False\}$ **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 $Value_i = [z \backslash in\ Proc\_Subset\ |-> Value_{i,z}]$ **then**

    // with $Type(Value_{i,z}) \in \{Integer, Boolean, String\}$

    $Translation(x_i = Value_i) := x_i(z) = Value_{i,z}$

    George: Will consider multi-dim functions later

  else if $x_i \in \text{Finite\_set}_i := \{e_1, \dots, e_p\}$ **then**

    // s.th any element $Type(e_i)$ is in $\{Integer, Boolean, String\}$

$$Translation(x_i \backslash in \text{ Finite\_set}_i) := x_i = e\_1 || \dots || x_i = e\_p$$

The translation of Init is:

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

Translating Next

// Recall that Next == $\bigvee_{i=1}^{k} (Predicate_i \bigwedge (x_i' = Value_i))$

for $i = 1$ **to** k **do**

  Define a Cubicle-transition T_i as follows:

Algorithm 1 TLA++ - Cubicle Translator

Defining Transition T_i

(13) Compute $Translation(Predicate_i)$ as in Section 1.2 and place it in T_i "requires"-part

(14) **if** $Type(Value_i) \in \{Integer, Boolean\}$ **then**

(15) $Translation(x_i' = Value_i) := (x_i := Value_i)$

else if $Value_i = "Text_i"$ **then**

(16) $Translation(x_i = Value_i) := (x_i =: Text_i)$

else if $[Value_{old}\ EXCEPT\ ![c] = e\ ]$ **then**

(17)
```
x(z) = case |  z = c : e
            |_ : Value_{old}[z]
```

(18) The translation of Next is:

(19)
```
transition Ti (z)
requires {
            Translation(Predicate_i)   }
          {
          Translation(xj1=Valuej1);     ;
            ...
            ...
          Translation(xj1=Valuejh);
          }
```

Translating invariant

// Recall that $Invar\_k == \backslash E\ Obj\ \backslash in\ Proc : P(Obj)$,

// where P is a proposition respecting the rules of Sec 1.2

(20) George: next step: considering invariants with $Obj1, Obj2inProc$

(22) Translate the negation of each invariant $Invar_1, \ldots, Invar_s$ as in Section 1.2 ;

(24) 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 $T_1(z), \ldots, T_k(z)$.