Understanding the TLA+ syntax

Dr. Tianxiang Lu

June 01, 2015

1 Constant Operators

1.1 Logic

First of all, TLA$^+$ inherits the syntax and semantics of standard first-order logic. Table 1 summarizes the syntax and the meaning. Here all the occurrences of x are bound in the expression p. TLA$^+$ requires that an identifier be declared or defined before it is used, and that it cannot be reused, even as a bound variable, in its scope of validity.

BOOLEAN	$\{\text{TRUE}, \text{FALSE}\}$
$\wedge, \vee, \neg, \Rightarrow$	and, or, not, implies
$\forall x : p$	for all interpretation of x, formula p holds
$\exists x : p$	there exists an interpretation of x, s.t. formula p holds
$\forall x \in S : p$	for all element x of set S, formula p holds
$\exists x \in S : p$	there exists an element x of set S, s.t. formula p holds

Table 1: Syntax of First-Order Logic in TLA$^+$

1.2 Sets

Elementary set theory is based on a signature that consists of a single binary predicate symbol $\in$ and no function symbols. Nevertheless, TLA$^+$ provides some syntactic sugar for set expressions, operations and comparisons as summarized in Table 2.

$=, \neq, \subseteq$	equal, not equal, subset or equal
$\in, \notin$	a member of, not a member of
$\cup, \cap, \setminus$	union, intersection, set difference
$\{e_1, \ldots, e_n\}$	Set consisting of elements e_i
$\{x \in S : p\}$	Set of elements x in S satisfying p
$\{e : x \in S\}$	Set of elements e such that x in S
SUBSET S	Set of subsets (the power set) of S

Table 2: Syntax of Set Theory in TLA$^+$

For a finite set S, TLA$^+$ provides a standard module introducing the expression $Cardinality(S)$ to denote the number of elements in S. The expression $Cardinality(S)$ is abbreviated to $|S|$ in our lecture.

1.3 Naturals and Arithmetic

There are several standard modules written in TLA$^+$ to provide the primitives of common modeling tasks. The standard module *Naturals* of the TLA$^+$ library provides operators representing natural numbers and arithmetic operators.

Table 3 summarizes the symbols and definitions in these modules. Here the variables a and b are natural numbers.

$a + b$, $a - b$, $a * b$, a^b	binary operators plus, minus, multiplication and exponents
$a < b$, $a \leq b$, $a > b$, $a \geq b$	binary comparisons
$a..b$	$\{n \in \mathbb{N} : a \leq n \leq b\}$
$a\%b$	a mod b, such that $0 \leq a\%b < b$
$a \div b$	division, such that $a = b * (a \div b) + (a\%b)$

Table 3: Syntax of Arithmetic in TLA$^+$

Since the symbol $\div$ is defined for *Naturals* and the symbol / (divisions) is defined for *Reals* in TLA$^+$, our lecture uses only $\div$ for division as defined in Table 3.

1.4 The Choose Operator

TLA$^+$ heavily relies on Hilbert's choice operator. An expression CHOOSE $x : p$ means taking some arbitrary value x that makes the expression p TRUE, or in other words, satisfies the property p. If no such value exists, CHOOSE will simply pick an arbitrary unspecified value. In practice, reasoning about expressions involving CHOOSE requires proving that there exists some x satisfying p. Note that all occurrences of x in the term p are bound. As syntactic sugar, the TLA$^+$ expression CHOOSE $x \in S : p$ means CHOOSE $x : (x \in S) \wedge p$, which means take value x, which is at the same time an element of S and satisfies the property p, if such a value exists.

1.5 Functions

Besides set and logic operations, functions are a convenient way to represent different kinds of data structures. In contrast to certain traditional constructions of functions within set theory, which construct functions as special kinds of ordered pairs, TLA$^+$ assumes functions to be primitive and assumes tuples to be a particular kind of function.

The application of function f to an expression e is written as $f[e]$. The set of functions whose domain equals X and whose co-domain is a subset of Y is written as $[X \to Y]$. The expression $x \in$ DOMAIN f denotes that the element x is an instance of the domain of the function f. The expression $[x \in X \mapsto e]$ denotes the function with domain X that maps any $x \in X$ to e; again, the variable x must not occur in e and is bounded to X. In fact, this expression can be understood as the TLA$^+$ syntax for a lambda expression $\lambda x \in X : e$.

TLA$^+$ introduces the notation: $[f$ EXCEPT $![e_1] = e_2]$. It means that the resulting function, say f_{new}, is equal to the function f except at the point e_1, where its value is replaced with e_2, namely $f_{new}[e_1] = e_2$. Here both e_1 and e_2 are expressions.

1.6 Tuples

A tuple is a function with domain $1..n$, for some natural number n. Hence, the expression $\langle e_1, \ldots, e_n \rangle$ denotes the n-tuple whose i^{th} component is e_i. The function application $e[i]$ here denotes the i^{th} component of tuple e. The cross product of n sets $S_1 \times \cdots \times S_n$ represents the set of all tuples with i^{th} component in S_i.

1.7 String and Records

TLA$^+$ provides records to facilitate access to components of tuples by giving them "names" as strings. In fact, strings are defined as tuples of characters and records are represented as functions whose domains are finite sets of strings. Since tuples, strings and records are all based on functions, the update operation on functions can be applied to them as well. Furthermore, TLA$^+$ offers more abbreviations for record operations. $e.h$ denotes the h-component of record e. $[h_1 \mapsto e_i, \ldots, h_n \mapsto e_n]$ denotes the record whose h_i component is e_i. $[h_1 \in S_1, \ldots, h_n \in S_n]$ denotes the set of records with field names h_i and whose respective values are in S_i.

2 Modules

For modeling large systems, specifications in TLA$^+$ can be organized in separate *modules*. A *module* is a basic unit of a TLA$^+$ specification, which typically contains *declarations*, *definitions* and *assertions*.

In TLA$^+$, an identifier must be declared or defined before it is used, and it cannot be reused, even as a bound variable, in its scope of validity. Every symbol must either be a built-in operator of TLA$^+$ (like $\in$) or it must be declared or defined. The scope of its validity is normally within the module, say M_1, where it is defined, but it can be extended to other modules, say M_2, by declaration of that module using statements of the form EXTENDS M_1 in the module M_2. A module could also be an instantiation of another module, but this feature is not used in our lecture. Form and details can be found in [**?**].

2.1 Declarations

A module may extend other modules, importing all their declarations and definitions, constant parameters and variables. Constant parameters represent entities whose values are fixed during system execution, although they are not defined in the module because they may change from one system instance to the next. Note that there are no type declarations because TLA$^+$ is based on set theory and all values are sets.

Variable parameters represent entities whose values may change during system execution. They correspond to program variables in this sense.

A specification of a distributed system can be modularized to static and dynamic modules. The static modules declare the constant parameters as primitive data structures. Then these static modules are extended to the dynamic modules where variable parameters are declared.

2.2 Definitions

In TLA$^+$, definitions are given in the form:

$$Op(arg_1, \ldots, arg_n) \triangleq exp$$

Here, Op is defined to be the operator such that $Op(e_1, \ldots, e_n)$ equals exp, where each arg_i is replaced by e_i. In case no arguments are given, i.e. $n = 0$, it is written as $Op \triangleq exp$. This shows the essence of a definition: it assigns an abbreviation (syntactic sugar, synonym) to an expression, which will never change its meaning (semantics) in any context. For example, $x \triangleq a + b$ means that x is syntactic sugar for "sum of a and b". Therefore, the expression $x * c$ is equal to $(a + b) * c$, instead of $a + b * c$ $(= a + (b * c))$.

For modeling distributed systems, complex data structures and their operators are typically introduced using definitions without or with arguments respectively. Note that argument arg_i is declared locally within the definition and its scope is only within exp.

Although definitions are used to define system operations in a way which appears similar to common programming languages, these operations have no "side effects", they are all functional. In TLA$^+$, a symbol or an identifier can only be declared or defined once. From then on, it will never change its meaning. Therefore, TLA$^+$ language is more assertional than operational. From the programming language perspective, it is declarative but not imperative.

2.3 Assertions

In TLA$^+$, assertions include assumptions, lemmas and theorems. For modeling distributed systems, assumptions can be either explicitly stated or implicitly encoded as preconditions of the actions. The explicit way helps the model checker to constrain the number of explorable states. The implicit way helps to give a more plausible implementation of the real system. Therefore, we try to avoid using explicit assertions for assumptions.

Lemmas and theorems typically assert causal relationships between definitions. There is no formal distinction in TLA$^+$ between a system specification and a property: both are expressed as formulas of temporal logic. Hence, asserting that specification S has property F amounts to claiming validity of the implication $S \Rightarrow F$. In our lecture, lemmas and theorems are typically written in this form. The proof language of TLA$^+$ also introduces the notion of *sequent* for stating assertions, which will be introduced later.

3 Other TLA$^+$ Syntax

TLA$^+$ uses syntactic sugar to extend the primitive functionality, which facilitates the readability, in particular for software engineers who are used to conventional programming language.

$$\text{IF } p \text{ THEN } e_1 \text{ ELSE } e_2$$

The expression is equivalent to $(p \land e_1) \lor (\neg p \land e_2)$

$$\text{LET } d_1 \triangleq e_1 \ldots d_n \triangleq e_n \text{ IN } e$$

The expression LET defines local operators, while IN encloses the region where these are used.

$$\begin{array}{ll} \land\ p_1 & \text{conjunctions and disjunctions} \lor\ p_1 \\ \land\ \vdots & \lor\ \vdots \\ \land\ p_n & \lor\ p_n \end{array}$$

Conjunctions and disjunctions in TLA$^+$ typically start with a conjunction symbol or disjunction symbol, which is in fact cruft in a conventional logical expression. But this kind of expression gives a better overview of the hierarchical structure of the logical formula. For example, instead of writing $(d_1 \lor (b_1 \land b_2)) \land c_1 \land c_2$, it is preferable in TLA$^+$ to use the following expression:

$$\begin{array}{l} \land \lor d_1 \\ \quad \lor \land b_1 \\ \quad\quad \land b_2 \\ \land c_1 \\ \land c_2 \end{array}$$

4 Transition System

A distributed system is typically specified as a transition system with initial state $Init$ and its transitions, which are called *actions* in TLA$^+$.

4.1 Variables

The variables for modeling a distributed system are typically defined as functions whose domain are the processes (or nodes) of the system and values are the local instances of specific data structures. Recall that TLA$^+$ does not have types. Type invariants are typically introduced with proofs. They normally have the following form:

$$\begin{array}{rl} TypeCorrectness \triangleq & \land\ var_1 \in [Node \to S_1] \\ & \land\ \ldots \\ & \land\ var_n \in [Node \to S_n] \end{array}$$

Here var_i is the variable name such that $vars = \langle var_1, \ldots, var_n \rangle$. $Node$ is typically the data structure of distributed entities in the system. S_i is the set of particular data structures.

4.2 Different Levels of Formulas

In TLA$^+$, formulas have different levels. A *constant* formula describes the properties of constants and constant parameters but they do not contain state variables. For example, $I \in \text{SUBSET } Nat$ is a constant formula assuming that I does not contain state variables, which expresses that the data structure I is a subset of the natural numbers.

A *state* formula describes the properties of state variables. For example, $x \in I$ (where x is a state variable) is a state formula expressing that this variable is an instance of the data structure I.

An *action* formula describes the changes of state variables after a transition. For example, $x' = x + 1$ is an action formula expressing that the state variable x is incremented by 1 after this transition.

In an action formula, the primed variable refers to the variable after change and the unprimed variable refers to its original value.

A *temporal* formula describes the properties of state variables throughout the sequence of executions of transitional actions. For example, $\Box(x \in I)$ forms a temporal formula out of a state formula, which states that x is always a member of I. $\Box[x \in I]_e$ forms a temporal formula out of an action formula, stating that no matter how the actions are executed, the value of state variable x remains an instance of I. The index e is an expression stating the set of stuttering variables (whose values are not changed by transition) if no actions are executed. Typically the e is a sequence of variables $vars$ such that $vars = \langle var_1, \ldots, var_n \rangle$.

4.3 Initial States

The formula $Init$ consists of the initial assignments of all variables, and typically takes the following form.

$$\begin{aligned} Init \triangleq & \land var_1 = [d \in Node \mapsto v_1] \\ & \land \ldots \\ & \land var_n = [d \in Node \mapsto v_n] \end{aligned}$$

Here the var_i is the variable name such that $vars = \langle var_1, \ldots, var_n \rangle$. The expression v_i in each assignment is the initial value for the particular variable.

4.4 Transition Rules as Actions

The formula $Next$ defines the transition rule, which is typically a disjunction of all the actions. Each action formula is a first-order formula containing unprimed as well as primed occurrences of the state variables, which refer respectively to the values of these variables in the states before and after the action.

An action for modeling distributed systems in TLA$^+$ typically has the form:

$$\begin{aligned} ActionName(arg_1, \ldots, arg_m) \triangleq & \land precondition \\ & \land var'_1 = [var_1 \text{ EXCEPT } ![arg_i] = v_1] \\ & \land \vdots \\ & \land var'_k = [var_n \text{ EXCEPT } ![arg_i] = v_k] \\ & \land \text{UNCHANGED } \langle var_{k+1}, \ldots, var_n \rangle \end{aligned}$$

An action may or may not have arguments which can be used only in the scope of its definition. An action typically consists of conjunctions of formulas which have different purposes. The formula *preconditions* can be a conjunction of several formulas, which serve as preconditions for the action, or so-called enabling conditions.

The formulas below the enabling conditions are the post-conditions of the actions, which define how state variables (var_i, i from 1 to k) of the transition system are altered using the update mechanism of function introduced before. Note that the value of the variable is a function mapping each element of the domain to a certain value. The mechanism changes the variables of only one element, arg_i, which is a parameter of the action definition. The expression v_i is the new functional value of that variable assigned to that particular element arg_i.

Suppose the total number of variables is n, then the rest of the variables, which are not modified by this action, should be summarized in UNCHANGED $\langle var_{k+1}, \ldots, var_n \rangle$, which is shorthand for the formula $\langle var_{k+1}, \ldots, var_n \rangle' = \langle var_{k+1}, \ldots, var_n \rangle$.

Note that making subsequent changes to a state variable in an action logically means that this variable is equal to different values, which is obviously a FALSE statement. A FALSE formula in an action which is a conjunction means that this action is in fact by definition FALSE and therefore will never be enabled. For this reason, modifications of one variable are always defined using one statement as shown in the formula above.

4.5 Temporal Formulas

A specification of the system can be written in the form:

$$Init \land \Box[Next]_{vars}$$

This form of system specification is sufficient for specifying the safety part of a system. The symbol $\Box$ is a temporal operator meaning that the formula following it is "always" true, i.e. in all states of an infinite sequence of states. Here the overall formula means that all runs start with a state that satisfies the initial condition $Init$, and that every transition either does not change $vars$ (defined as the tuple of all state variables: $vars \triangleq \langle var_1, \ldots, var_n \rangle$) or corresponds to a system transition as defined by $Next$. In general, the index f of a formula $[N]_f$ can be any state formula, not just a tuple of variables. This is important for refinement proofs but not used in our lecture.