The Module Structure of TLA+

Leslie Lamport

12 September 1996

Back to TLA home page

This note informally describes the syntax and semantics of TLA+ down to, but not including, the level of expressions. Its main purpose is to introduce and explain the constructs for including modules that were introduced in August, 1996.

Table of Contents

Modules, Declarations, and Definitions

A TLA+ specification is organized as a collection of modules. A module looks like this: (Note 1)

```
---------- MODULE DirectedGraphs ------------
 body
-----------------------------------------------
 more body
===============================================
```

The body consists of a sequence of statements, where a statement is a declaration, definition, assumption, or theorem. We will ignore assumptions and theorems for now. Horizontal lines like

```
-----------------------------------------------
```

can appear between statements; they are purely decorative.

A declaration statement looks like this:

```
CONSTANTS Node, Edge
```

It adds to the module the declarations of `Node` and `Edge` as constant symbols. TLA+ also allows variable symbols, identified by the keyword "`VARIABLES`". (Note 2) A declared symbol is a "free parameter" of the module.

A definition statement looks like this:

```
NonEdge == (Node \X Node) \ Edge
```

Assuming that `Edge` and `Node` are declared as above, this statements adds to the module the definition that defines the symbol `NonEdge` to equal the expression

```
(Node \X Node) \ Edge
```

We will not worry about the meaning of this and other expressions.

We can follow this definition statement with the definition statement

```
NonTrivialNonEdge == NonEdge \ {<<n, n>> : n \in Node}
```

This defines the symbol `NonTrivialNonEdge` to be the expression

```
((Node \X Node) \ Edge) \ {<<$n, $n>> : $n \in Node}
```

Observe that to obtain this definition, we replace the defined symbol `NonEdge` by its definition, and we replace the bound variable `n` with a new symbol `$n` that cannot be declared or defined by the user.

In this way, a definition always defines a symbol to equal an expression containing only declared symbols, bound variables that are different from any symbols typed by the user, and the built-in operators of TLA+ like `\X` and `\`.

Symbols can also be defined to equal operators that take arguments. (Note 3) For example,

```
Nbrs(n, m) == <<n, m>> \in Edge
```

defines the symbol `Nbrs` to be an operator that assigns to any expressions `exp_1` and `exp_2` the expression

```
<<exp_1, exp_2>> \in  Edge
```

We will denote this operator

```
LAMBDA $n, $m : <<$n, $m>> \in \ Edge
```

We use `LAMBDA` only to describe the semantics of modules; it is not an operator of TLA+. (As before, we replaced the bound variables `n` and `m` by "untypable" symbols `$n` and `$m`.) (Note 4)

TLA+ allows you to define prefix, infix, and postfix operators. For example, writing

```
n ?? m == <<n, m>> \in Edge
```

defines the infix operator `??` to be the same as the ordinary operator `Nbrs`. (Note 5)

In a definition, the expression to the right of the "==" can contain only TLA+ primitives, symbols that have already been defined or declared, and definition parameters (for example, the symbols `n` and `m` in the definition of `Nbrs`). Circular definitions are not allowed. (Recursive definitions will be discussed elsewhere.)

Including One Module in Another

One builds large hierarchical specifications by building a new module "on top of" old modules. One way to do this is with the `EXTENDS` statement at the beginning of the module. The statement

```
EXTENDS Foo, Bar
```

simply adds the declarations and definitions from modules `Foo` and `Bar` (Note 6) to the current module. This has approximately (Note 7) the same effect as inserting the body of those modules in the current module.

The other way to use an existing module is with the `INSTANCE` statement, which instantiates (substitutes for) the module's declared symbols. Suppose that module `DirectedGraphs` declares the symbols `Node` and `Edge`, and defines the symbols `NonEdge` and `Nbrs`, as above. Consider the following module:

```
------------------ MODULE SGraphs ------------------
CONSTANT S
```

```
Edge == { <<m, n>> \in S \X S : m \in n }

INSTANCE DirectedGraphs WITH Node <- S, Edge <- Edge
=======================================================
```

The INSTANCE statement adds to module SGraphs all definitions obtained from the definitions of DirectedGrahs by substituting S for the symbol Node and substituting the definition of Edge (from module SGraphs) for the symbol Edge. Thus, the INSTANCE statement adds the following definitions to module SGraphs:

- NonEdge is defined to equal

```
(S \X S) \ { <<$m, $n>> \in S \X S : $m \in $n }
```

- Nbrs is defined to equal

```
LAMBDA $n, $m : <<$n, $m>> \in
                  { <<$m, $n>> \in S \X S : $m \in $n }
```

Substitutions such as Edge <- Edge, in which a symbol is substituted for itself, are quite common. We therefore introduce the convention that if no substitution for a declared symbol s of an instantiated module appears in the WITH clause, then there is an implicit s <- s. (This means that s must already be declared or defined in the current module.) Thus, the INSTANCE statement of module SGraphs above can be written

```
INSTANCE DirectedGraphs WITH Node <- S
```

If there are no explicit substitutions, the "WITH" is omitted.

A module may need to use more than one instantiation of the same module. For example, we may want to use two different instances of the DirectedGraphs module, with different substitutions for the parameters Node and Edge. We can't let the same symbol have two different definitions, so we must rename the defined symbols. The statement

```
Foo == INSTANCE DirectedGraphs WITH Node <- S
```

is the same as the INSTANCE statement in module SGraphs above except that it defines the symbols Foo.NonEdge and Foo.Nbrs instead of the symbols NonEdge and Nbrs. (Note 8)

Sometimes we need to use a whole family of instantiations of the same module. We do this as indicated by the following example:

```
E(Set) == { <<T1, T2>> \in (SUBSET Set) \X (SUBSET Set)
              : T1 \subseteq T2 }

DG(S) == INSTANCE DirectedGraphs
           WITH Node <- SUBSET S, Edge <- E(S)
```

(T1 and T2 are bound variables in the first definition.) For any expression exp, this defines DG(exp).NonEdge to equal the expression obtained from the definition of NonEdge in the DirectedGraphs module by substituting SUBSET S for Node and substituting the definition of E(exp) for Edge. In other words, DG(exp).NonEdge equals

```
((SUBSET exp) \X (SUBSET exp))
   \ { <<$T1, $T2>> \in (SUBSET exp) \X (SUBSET exp)
       : $T1 \subseteq $T2 }
```

We can think of DG(exp).NonEdge as meaning DG.NonEdge(exp), where DG.NonEdge is defined to be

```
LAMBDA $S : ((SUBSET $S) \X (SUBSET $S)) \
              { <<$T1, $T2>> \in (SUBSET $S) \X (SUBSET $S)
                : $T1 \subseteq $T2 }
```

The INSTANCE statement above similarly defines DG(exp_1).Nbrs(exp_2,exp_3) for any expressions exp_1, exp_2, and exp_3. Again, we can think of DG(exp_1).Nbrs(exp_2,exp_3) as meaning DG.Nbrs(exp_1)(exp_2,exp_3), where DG.Nbrs is defined to be

```
LAMBDA $S : LAMBDA $n, $m :
   <<$n, $m>> \in { <<$T1, $T2>> \in (SUBSET $S) \X (SUBSET $S)
                    : $T1 \subseteq $T2 }
```

To summarize:

`EXTENDS M`
: Adds to the current module the declarations and definitions from module M.

`INSTANCE M WITH s_1 <- exp_1, ... , s_k <- exp_k`
: The s_i must be the declared symbols of module M. (If k=0, we omit the "WITH". Omitting an explicit instantiation of a declared symbol s from the WITH clause is equivalent to adding the instantiation s <- s.) This INSTANCE expression represents a mapping that assigns to each defined symbol ds a definition ds_def, defined as follows. Let EXP_i be the expression obtained from exp_i by replacing every symbol defined in the current module with its definition, and replacing bound variables by new symbols. We let ds_def equal the expression obtained from the definition of ds in M by replacing each declared symbol s_i with EXP_i.

  - If this instance expression appears as a statement by itself, then each defined symbol ds of M is defined in the current module to equal ds_def.
  - If this instance expression appears in the statement IM(p_1, ..., p_j) == INSTANCE..., then IM.ds is defined to equal

    ```
    LAMBDA $p_1, ... , $p_j : DS_def
    ```

    for each defined symbol ds of M, where DS_def is obtained from ds_def by replacing each symbol p_i by the new symbol $p_i. We write IM(e_1, ... , e_j).ds instead of IM.ds(e_1, ... , e_j). If j=0, we write IM ==... and IM.ds instead of IM()==... and IM().ds.

The keyword INSTANCE can be followed by a comma-separated list of instantiations, as in

```
INSTANCE Naturals, Sequences
```

If a module M has no declared symbols, then EXTENDS M and INSTANCE M are equivalent. EXTENDS should be used only for breaking up what is logically a single specification into smaller pieces. If M is a general-purpose module, INSTANCE M should be used.

Modules Inside Modules

One can put one module inside another, as in

```
------------- MODULE Outer ------------
CONSTANT z
...
   ----------- MODULE IMod -----------
   CONSTANT x, y
   ...
   ====================================

Inner(x,y) == INSTANCE IMod
...
```

Module IMod is like any module, except that it can use the declared symbol z and any other symbols declared or defined before it in module Outer. Module IMod may be used only inside module Outer. (Note 9)

Submodules such as IMod are often used in the following way. Suppose we want to define

```
Spec == \exists x, y : InnerSpec
```

where InnerSpec is some complicated formula involving x and y. The obvious way to do this is:

```
CONSTANT x, y
...
InnerSpec == ...
Spec == \exists x, y : InnerSpec
```

However, the definition of Spec is illegal because it violates the following syntactic rule of TLA+:

- A symbol that is already declared or defined cannot be used as a bound variable.

(The purpose of this rule is to prevent confusion between bound variables and defined or declared symbols.) We could write

```
InnerSpec(x, y) == ...
Spec == \exists x, y : InnerSpec(x,y)
```

However, InnerSpec may be defined in terms of a sequence of other defined symbols, each of which is defined in terms of x and y. We would have to make x and y explicit parameters of all those definitions. Instead, using the module structure in the example above, we can put the definition of InnerSpec in submodule IMod and define

```
Spec == \exists x, y : Inner(x, y).InnerSpec
```

Local Definitions

It often happens that a module A extends modules B and C, both of which extend module D. For example, D might declare symbols and make some definitions that are used by both B and C. Module A then obtains two copies of the definitions and declarations of D--one obtained by extending B and the other obtained by extending C. TLA+ allows the same symbol to be declared or defined two or more times, if the declarations or definitions are the same. (Note 10)

Adding more and more definitions with each level of module inclusion will eventually lead to unintended name clashes. We don't want to know what modules are included by Naturals (the module that defines the natural numbers and operators like +), so we need a way to avoid name clashes with symbols defined by those modules. TLA+ provides local definitions as a way of avoiding such name clashes. Preceding a definition with the keyword LOCAL makes that definition local to the current module. For example, writing

```
LOCAL Temp(x) == x + y
```

defines Temp in the current module. But the definition of Temp is not obtained by a module that includes the current module--whether with EXTENDS or INSTANCE. Similarly, writing

```
LOCAL INSTANCE M WITH ...
```

or

```
LOCAL Temp(x) == INSTANCE M WITH ...
```

makes all the definitions included from M local to the current module.

Declarations cannot be made local. Symbols are defined in terms of declared symbols, and it wouldn't make sense to make a symbol local if it appeared in a nonlocal definition. (Remember that the statement df == exp defines df to be the expression obtained from exp by replacing defined symbols with their definitions, so the definition of df has no defined symbols.) Hence, one cannot precede a declaration or an EXTENDS statement with "LOCAL". (Remember that

`EXTENDS M` adds `M`'s declared symbols to those of the current module. If `M` declares no symbols, one can write `INSTANCE M` instead of `EXTENDS M`.)

When writing a general-purpose module, definitions included from other modules should be made local. For example, a general-purpose `Graphs` module should include the `Naturals` module by

```
LOCAL INSTANCE Naturals
```

That way, a module that includes `Graphs` can define `+` to mean something other than addition of natural numbers.

Higher-Order Operators

Operators can take operators as arguments. For example

```
Double(A, F(_,_)) == F(A,A)
```

defines `Double` to be an operator that takes two arguments---the first of which is an expression, and the second of which is an operator that takes two arguments, both of which are expressions. For example, `Double(3,+)` equals `3+3`, and `Double({a},Nbrs)` equals `Nbrs({a},{a})`.

TLA+ does not allow an operator to take as an argument an operator that takes as an argument an operator. In other words, there can be at most two levels of parentheses to the left of the "==". One cannot define an operator that is any "higher-order" than `Double`.

The value of a TLA+ expression cannot be an operator; it must be a simple value. (A `LAMBDA` expression is one whose value is an operator, but one cannot write `LAMBDA` expressions in TLA+.)

One can declare constant symbols to be operators. For example,

```
CONSTANT Foo(_, _)
```

declares `Foo` to be an operator that takes two arguments. Since the value of an expression cannot be an operator, the declared operator `Foo` can be instantiated only by an operator that has the same number of arguments---for example, by writing

```
INSTANCE ... WITH Foo <- +
```

One cannot declare variable symbols to be operators.

Assumptions and Theorems

A module can contain assumptions and theorems, of the form `ASSUME exp` and `THEOREM exp`, where `exp` is an expression. The expression `exp` can contain symbols declared or defined anywhere in the module.

For simplicity, suppose that module `M` contains only the single assumption `ASSUME a` and the single theorem `THEOREM t`. (The generalization is obvious.) Let `A` and `T` be the expressions obtained from `a` and `t` by replacing all defined symbols with their definitions, so the free variables of `A` and `T` are declared symbols. We make the syntactic requirement that `A` contain only constant symbols. (Note 11) Module `M` is *valid* iff the formula `A => T` is valid. Writing a module asserts that the module is valid, so a theorem represents a proof obligation for the writer of the module.

When module `M` is included with an `EXTENDS M` statement, its assumptions and theorems are added to those of the current module. When module `M` is instantiated with an `INSTANCE M` expression, only its definitions are added to the current module, not its assumptions or theorems (or declared symbols).

Suppose `M` is valid, so `A => T` is a valid formula. Suppose that the instantiation

```
INSTANCE M WITH s_1 <- exp_1, ...  , s_k <- exp_k
```

is syntactically correct, meaning that every declared symbol of `M` is instantiated with an expression all of whose symbols are defined or declared. Substitution preserves validity, so the formula obtained from `A => T` by replacing each `s_i` with `exp_i` is valid.

One often write modules with assumptions but no theorems. These assumptions serve no logical function, since they are not used to prove anything (except perhaps if the module is extended by another module). However, they serve as useful comments to the reader about the values one expects these symbols to assume. For example, one might add to module `DirectedGraphs` the assumption

```
ASSUME  Edge \subseteq Node \X Node
```

which indicates that one expects `Edge` to be a set of ordered pairs of elements from the set `Node`. We expect the assumption to be satisfied when the module is instantiated, but we do not make this a formal requirement. (Note 12)

Expression Levels

TLA extends "ordinary math" (first-order logic plus set theory) with declared variable symbols and a few nonconstant operators. The nonconstant operators of TLA+ are

```
~> (leads-to)                   ' (prime)
-+-> (while)                    ENABLED
\cdot (action composition)      UNCHANGED
[...]_   (e.g., [A]_f)          \EE (temporal \exists)
<<...>>_ (e.g., <<A>>_f)        \AA (temporal \forall)
[]     <>     WF     SF
```

To define the semantics of TLA+, we take the following six to be primitive and we define the rest in terms of them:

```
'    ENABLED    \cdot    []    -+->    \EE
```

TLA places certain syntactic restrictions on the use of these operators. These restrictions are described in terms of *levels*. Declared symbols and primitive TLA+ operators are assigned one of four levels, listed below in increasing order:

Constant Level
: Contains declared constant symbols and constant-level TLA+ operators.

State Level
: Contains declared variable symbols and the `ENABLED` operator.

Action Level
: Contains the operators `'` and `\cdot`.

Temporal Level
: Contains the operators `[]`, `-+->`, and `\EE`.

The level of an expression is defined to be the level of its highest-level subexpression, except that `ENABLED A` is a state-level expression even when `A` is an action-level expression. Note that in determining the level of a subexpression, bound variables are considered to have constant level except that bound variables introduced by the temporal quantifiers `\EE` and `\AA` are state-level symbols.

TLA places the following syntactic restrictions on expressions.

- The expression `exp'` is allowed only if `exp` is a state-level or contant-level expression.

- A temporal-level expression is either
  - a constant-level or temporal-level operator applied to constant-level, state-level, or temporal-level expressions.

  - an expression having one of the the following forms

    ```
    [][A]_f     <><<A>>_f     WF_f(A)     SF_f(A)
    ```

    where `A` has at most action level and `f` has at most state level.

These level restrictions place restrictions on the arguments of defined operators. For example, consider the operator `S` defined by:

```
S(U, V, x)  ==  U => Enabled (V \/ x')
```

The expression `S(e_1, e_2, e_3)` is legal iff expression `e_2` has at most action level, and `e_3` has at most state level. The level of the complete expression is the maximum of the level of `e_1` and state level. (Note 13)

A constant-level module is one that has no declared variable symbols and uses only constant-level operators. Constant-level modules are used for defining data structures and for describing "ordinary math".

We make the following rules for instantiation (by an `INSTANCE` expression):

- Except when instantiating a constant-level module, a constant symbol can be instantiated only by a constant-level expression. (Arbitrary instantiations of declared symbols are allowed when instantiating a constant-level module.)

- Variable symbols can be instantiated only by constant-level or state-level expressions.

These restrictions ensure that the instantiated modules satisfy the syntactic level restrictions of TLA. They are also needed to ensure that the instantiation of a valid module is valid. For example, the TLA formula `[][c'=c]_c` is valid (true for all behaviors) if `c` is a declared constant symbol. Substituting a variable for `c` would produce an invalid formula (one that is false for some behaviors).

A Closer Look at Instantiation

We have stated that instantiation preserves validity. If `F` is defined in module `M` to equal a valid formula, and we write

```
I == INSTANCE M WITH ...
```

then `I.F` is a valid formula. Instantiation is substitution, and in the presence of quantifiers, substitution preserves validity only if it is defined properly. In particular, substitution must be defined so capture of free variables by quantifiers is not allowed. For example, consider the valid formula `\exists u : u /= v`. Naive substitution of `u` for `v` in this formula would yield the invalid formula `\exists u : u /= u`. Validity is not preserved because the free symbol `u`, which is being substituted for `v`, is captured by the quantifier `\exists`.

The problem of capture of free variables by explicit quantifiers is avoided in TLA+ by renaming bound variables. The module

```
---------------- MODULE M ----------------
CONSTANT v
F == \exists u : u /= v
==========================================
```

defines F to be the formula \exists $u : $u /= v. The statement

```
I == INSTANCE M WITH v <- u
```

then defines I.F to be the valid formula \exists $u : $u /= u.

Some nonconstant operators of TLA+ contain implicit quantification--most notably, the operator ENABLED. Suppose module M is

```
---------------- MODULE M ----------------
VARIABLE u, v
F == ENABLED ((u' = u) /\ (v' /= v))
==========================================
```

Then F is a valid formula. Now consider the statement

```
I == INSTANCE M WITH u <- x, v <- x
```

A naive substitution would make I.F the formula

```
ENABLED ((x' = x) /\ (x' /= x))
```

which is equivalent to FALSE. The problem is that, within an ENABLED expression, primed variables are really bound, so this naive substitution results in variable capture. The meaning of formula F is really

```
\exists $uprime, $vprime : ($uprime = u) /\ ($vprime = v)
```

To ensure that instantiation preserves validity, we make the following rule for the ENABLED operator:

- When instantiating a module, every declared variable symbol that occurs primed in an ENABLED expression is renamed to a new variable symbol before substituting for the declared symbols.

For example,

```
------------------ MODULE M ------------------
VARIABLE u
G(v, A) == ENABLED (A \/ ({u, v}' = {u, v}))
H == G(u, u' /= u)
============================================
```

defines G to equal

```
LAMBDA $v, $A : ENABLED ($A \/ ({u, $v}' = {u, $v}))
```

and defines H to equal

```
ENABLED (u' /= u) \/ ({u, u}' = {u, u})
```

Before instantiating M, the definition of G is changed to to

```
LAMBDA $v, $A : ENABLED ($A \/ ({$$u, $v}' = {u, $v}))
```

and the definition of H is changed to

```
ENABLED ($$u' /= u) \/ ({$$u, $$u}' = {$$u, $$u})
```

(The two "$"s in the name $$x are meant to indicate that it is a new bound variable symbol.) Note that the substitution for u' in the definition of G occurs *after* that definition is used in defining H.

As another example, consider the module

```
----------------- MODULE MM -----------------
VARIABLES u, v
```

```
A    ==  (u' = u) /\ (v' /= v)
B(d) ==  ENABLED d
C    ==  B(A)
============================================
```

and the instantiation

```
VARIABLE x
I == INSTANCE MM WITH u <- x, v <- x
```

This instantiation yields

```
I.A  =  (x' = x) /\ (x' /= x)
I.B  =  LAMBDA $d : ENABLED $d
I.C  =  ENABLED (($u' = x) /\ ($v' = x))
```

Note that `I.C` is not equivalent to `I.B(I.A)`. In fact, `I.C` is valid while `I.B(I.A)` is equivalent to `FALSE`.

The other TLA+ primitives that have implicit quantification are `\cdot` (action composition) and `-+->` (while). The rule for `\cdot` is similar to that for `ENABLED`, where in the expression `e_1 \cdot e_2`, primed variables in `e_1` and unprimed variables in `e_2` are bound. The rule for `-+->` is that, before instantiating a module, each instance of `P -+-> Q` is replaced by an equivalent formula containing explicit quantification. The precise formula can be found elsewhere.

Note 1

One can also write

```
BEGIN MODULE DirectedGraphs
 body
END
```

The precise ASCII syntax of module delimiters and decorative horizontal lines has not been determined yet.

Note 2

The "`S`" at the end of "`VARIABLES`" and "`CONSTANTS`" is optional. The keyword "`PARAMETER(S)`" means the same as "`CONSTANTS`". If you are using TLA+ for ordinary mathematics, with no actions or temporal formulas, then all the parameters are constants.

Note 3

Operators that take arguments are different from functions. Functions, and their definitions, will be discussed elsewhere.

Note 4

The precise rule for turning a definition statement into a definition is: First replace all defined symbols to the right of the == by their definitions; "beta-reduce" `LAMBDA` expressions when possible--for example, reduce

```
(LAMBDA $a : <<$a, $a>> \in S)(Y+Z)
```

to `<<Y+Z, Y+Z>> \in S`--and finally, replace all bound symbols, including the parameters of the definition (the symbols `n` and `m` in the definition of `Nbrs`) by "untypable" symbols that do not already appear in the expression.

Note 5

TLA+ provides large, fixed sets of infix and postfix operator symbols; they will be described elsewhere.

Note 6

The translation from module names to modules is outside the scope of this document. Presumably, a tool will have rules for finding named modules. We suggest that a module

named N appear in a file named N.tla. The directory in which this file is to be found will be system dependent.

Note 7

If module Foo makes local definitions that clash with definitions in the current module, then the body of Foo cannot be inserted into the current module. This is the only reason EXTEND Foo is not equivalent to inserting the body of Foo in the current module.

Note 8

If module DirectedGraphs had defined an infix operator ??, then this statement defines Foo.?? to be an infix operator. The expression a Foo.?? b looks strange, but the alternatives seem worse. In practice, one does not define infix operators in modules that one wants to use with this kind of renaming.

Note 9

Submodules provide a scoping mechanism for module names, which in principle could be used to define the mapping between module names and modules. For example, the modules written by user Jones of company XYZ could be submodules of a module named Jones, which is a submodule of a module named XYZ. However, in practice, such name scoping will be provided by directory structures and file search paths.

Note 10

Two declarations of a symbol s are the same if they both declare s to be a constant or both declare it to be a variable. Two definitions are the same if they have parse trees that differ only in the names of bound variables.

Note 11

In general, we should define validity of M to mean A |= T, which means that the validity of A implies the validity of T. If A is a constant formula, then A |= T is equivalent to the validity of A => T. For writing specifications, this special case is all we need, and restricting ourselves to it allows us to avoid introducing the semantic operator |=.

Note 12

It is tempting to include an instantiated module's assumptions as theorems, so they become proof obligations of the current module. However, it may be impossible to discharge those proof obligations in the current module. For example, suppose we were specifying a graph algorithm and we wrote

```
VARIABLES N, E
INSTANTIATE DirectedGraphs WITH Node <- N, Edge <- E
```

where N and E represent variables of the algorithm. The instantiated assumption of the DirectedGraphs module is E \subseteq N \X N. It is impossible to prove this formula because it is not valid; N and E are flexible variables, so they can assume any values. What we do expect to be able to prove is

```
(*)   Spec => [](E \subseteq N \X N)
```

where Spec is the TLA formula that specifies the algorithm. Formally, there is nothing special about Spec; it is just one of many defined symbols. So, there is no reason to take (*) as the proof obligation associated with the instantiation. Moreover, Spec might not even be defined in the current module, but in some other module that extends the current one.

Note 13

For simplicity, we always consider Enabled A to have state level, even in those special cases when we could determine that it is a constant formula.