Implementation of a compiler from Pluscal to TLA+ with Tom

Marc PINHEDE

Second year engineering school internship report
between 4th of June and 4th of August 2012
Supervisor : Martin QUINSON
## Contents

1 Institutional context 4  
   1.1 Laboratory .................................................. 4  
      1.1.1 General presentation .................................... 4  
      1.1.2 Organization ............................................. 5  
   1.2 Research team .............................................. 5  

2 Scientific context 7  
   2.1 Antlr .......................................................... 7  
   2.2 Tom .......................................................... 8  
   2.3 Tlc .......................................................... 9  

3 Pluscal2.0 10  
   3.1 Grammar redactions .......................................... 10  
   3.2 Tree transformations ......................................... 11  
   3.3 Code generator ............................................... 12  

4 Evaluation 13  
   4.1 Tests ........................................................ 13  
   4.2 Working exemples ............................................ 14  

A A simple compilation 17
Introduction

As internet grows and becomes more and more used, the number of computers and servers all over the world increases every day. Computers becoming more performants, people get used to create heavier programs, needing more calculation abilities. To permit this, and many others applications, we began developing distributed algorithms, using more machines for the same program, allowing by this way more physical ressources to a single task. However designing distributed programs is a difficult task. It is pretty easy to commit a mistake in those algorithms, because pretty hard to keep the entire system in mind. An unfortunate race condition, or a deadlock could compromise the system and cause the program to have an unexpected behaviour. As distributed algorithms cover nowadays many security fields, they must be reliable.

To ensure that reliability, one of the offered techniques is an algorithmic verification, the model checking, which builds a graph of all possible states and transitions, where states are labelled with properties. To be understood by the model checker, the system and the properties we want to check must then be defined in a very formal and mathematical way. To that extent, Leslie Lamport comes up with TLA+. This specification language uses actions’ descriptions to determine states and transition. But writing specification in TLA+ requires a good mathematic knowledge, or at least some habits in describing systems. Algorithm designers are usually not familiar with that way of considering algorithm. To simplify TLA+ programs writing, Lamport created another language: PlusCal. PlusCal is a higher level language with simple statements that permit to describe a system in a more classic programatic way.

Actually, the PlusCal language does not really reach its goals. Writing specification in PlusCal requires to understand at least TLA+, and maybe even the way Lamport’s program translate PlusCal in TLA+. The real PlusCal objective was to easily write the first draft in TLA+. Then the programmer could work on the generated TLA+ code to complete it and make it match his expectations. That is why the original compiler requires the PlusCal code to be written in comments at the top of a TLA+ file. Furthermore, PlusCal suffers from some others technical complications. It offers a way to describe processes but only at top level, with no hierarchical relations between them. It also gives a way to consider atomics statements, which are statements needing to be executed one after the others, without intervention from others processes. To describe atomic blocks, labels are used, so it’s impossible to place a label in an atomic block, which could be restrictive for the user.

Those limitations, and some others I will not list here, lead Sabina AKHTAR to validate a Thesis about a new language, PlusCal 2.0, based on PlusCal, but without any limitation she listed. Even if her thesis describes very well the procedure to translate PlusCal 2.0 in TLA+, the technical approach she chooses locks any possible changement or improvement on her work. The main objective of my internship was to reimplement this compiler with others tools, making it easier to modify.
Chapter 1

Institutional context

1.1 Laboratory

1.1.1 General presentation

LORIA is the French acronym for the Lorraine Research Laboratory in Computer Science and its Applications. It is a research unit (UMR 7503), common to:

- CNRS (Centre Nationale de Recherche Scientifique),
- University of Lorraine
- INRIA (Institut National de Recherche en Informatique et Automatique).

This unit was officially created in 1997.

LORIA’s missions mainly deal with fundamental and applied research in computer sciences. The lab is a member of the Charles Hermite Federation, which groups the four main research labs in Mathematics, in Information and Communication Sciences and in Control and Automation. Bolstered by the 500 peoples working in the lab, its scientific work is conducted in 27 teams including 16 common teams with INRIA. LORIA is today one of the biggest research labs in Lorraine.

LORIA was structured in five departments:

- algorithms: computation, image and geometry,
- formal methods,
- networks, systems and services
- knowledge and language management,
- complex systems and artificial intelligence.

The repartition per post of the 500 workers is:

- 168 full time researchers,
- 14 full time administrative support staff
- 130 PhD students,
- 105 post-doc,
- 50 engineers,
- 50 trainees.
1.1.2 Organization

The organization chart 1.1.2 show the hierarchy’s structure.

Direction team

This instance gathers the director of the LORIA, the two Deputy Directors and scientists representing the different major themes of the lab. Its role is to assist the director in decisions and their implementations.

Laboratory council

The laboratory council is composed in four-year elected members and appointed members. It is the only advisory instance of the lab where the whole staff is represented. It has an advisory role to the director for all questions concerning the research unit: scientific policy, laboratory organization, budget, recruitment, trainings, internal rules, management of working hours, work condition... The laboratory council meets at least once each quarter.

1.2 Research team

Presentation

My own intership happened in the AlGorille team, where Martin QUINSON works.

It is composed of 4 permanent member, 3 PhD students, 2 associated members and 4 engineers. Jens GUSTEDT is its leader.

The AlGorille team is working on distributed system. And more precisely on the notion of ‘grid’. The grid is close to the distributed system, but is the origin of more constraints: less knowledge from both, the user and the general environment, heterogeneous environments, a different scale.... The team has the following challenges: overcome spacial and temporal distance (latency problems, data consistency and data persistence problems), reconcile provider and customer on the allocated resources, and keep a good performance evaluation of applications.

AlGorille also have the responsibilities of the Grid’5000 Nancy’s cluster. Grid’5000 is an experimental plateform, reparted in 10 locations whose objectif is to experiment distributed system, and to provide researchers hardware to make large scale computation.

Research activities

- Structuring of applications for scalability
- Transparent resource management
- Experimental validation

Simgrid

Simgrid is a project leaded by Martin QUINSON (LORIA, University of Lorraine), Arnaud LEGRAND (CNRS Grenoble), Henri CASANOVA (University of Hawaii at Monoa) and Frederic SUTER. The project was started in 2000.

The aim is to provide a tool box which can be used to make simulation with distributed system. Simgrid has different interfaces depending on the work we want to do. It is delivered under LGPL licence so its source code is available on the net. The framework allows reproductible simulation, with a lot of tools to easily describe the experimentation. Simgrid is a powerful simulation framework and it is still improved by AlGorille team and the ones who are associated to.
Chapter 2
Scientific context

To achieve my task, Martin QUINSON, who supervise my internship gave me some leads to start. The lexer/parser could be generated by one of the classic tools available for that, like Antlr or Yacc. Then, following the thesis instructions, the first tree could be simplified in an intermediary language and translated into TLA+. For this task, a tool developed in the Loria, Tom, could perfectly fit our needs. To finish, a simple 'PrettyPrinter' could transform the tree in a String.

2.1 Antlr

The first part of any compiler program is the recognition of the input stream. That work is assigned to a lexer, which matches input characters or groups of characters and produces a stream of 'tokens'. Those tokens can be considered as characters or numbers which work like a code. Every token is corresponding to an element of the recognized language. The work of the lexer is therefore to substitute characters or integers for a correct word. Of course, if the lexer matches a wrong word, which does not belong to the language, an error is raised.

Then comes the Parser. The parser is responsible for checking that the input stream match the given grammar. So, when describing a language specification, like Pluscal2.0, we write a grammar file for antlr (.g) that contains the rules of the grammar. Antlr then generates a parser that checks that every given token is a viable alternative, considering the tokens it received before, and according to the grammar. That implies, of course, that the grammar should be LL. It globaly means that any input token is self-sufficient to find out which rule may be applied. In some cases, we can accept that we need two or more tokens to ensure that we choose the good rule. If the grammar is non-LL, or, in others words, if we need to see forward to know which rule of the grammar must be applied, antlr can use backtracking to clarify the meaning of the input stream. In this case, the generated parser will emit hypothesis, and try one of the possible rules for the given token. If it reaches a dead-end, it will comme back to the last done choice, and change it for an other. If every choices had been tested and no one is viable, an error is raised. Most of the time, the backtrack is not needed. If necessary, some changes can be brought to the grammar in order to avoid non-LL rules.

While parsing the input stream, the parser althought creates an implicit tree in memory. This tree only represents the way rules are called by each others. Every node is either a rule or a token. That tree contains enough information to find back the input stream. It also gives a good abstraction of the input program. But some parts of that tree are in fact useless. The priority of operators is a good exemple of constraint that leads to add many rules. Then, when an input program is parsed, all those rules are in the
generated tree. The basic way to avoid those useless nodes, is to rewrite a first time this
tree. We create an Abstract Syntaxic Tree (AST) from that first parsed tree. Antlr gives
us a very easy way to do that. In the grammar file, we can add some instructions which
permit a quick rewrite, even it is limited to the rule level and doesn’t permit rewriting
rule that implies two rules.

The choice of antlr was very good, because even if the syntaxe is really compact, it
generates big output file. Very few changes in the original antlr file can be responsible for
big changes in the generated file. The original antlr file is therefore a good format to work
with. We can also notice that one antlr file gives three steps of the compiling process,
and permit to save a lot of time.

2.2 Tom

Tom is a tool that is developed in the Loria laboratory, and is made to simplify the
use of data structures. This helpful tool can work with different languages, like java, c, ...
Technically speaking, tom’s code is included into an host language (here java). The
original file must then be parsed by tom’s compiler, that will change tom’s code into host
code. The file can then be parsed by the compiler of the host language because it doesn’t
include any more unknow code (here, javac). In the background, tom will manage itself
to create appropriate data structures. The user will just have to use tom expression to
work on tom objects. But any tom object is also a host object, so for our case, we can
use tom object as arguments of java functions.

Tom expressions are useful to work on tree configurations. To use Tom, we first have
to write a Tom ‘signature’. This signature represents the way Tom objects are linked to
each others and is written the same way that any classic grammar. Any Tom object can
have a certain amount of ‘son’ which can be tom objects or basic types (int, char,string,..). Any son, by the same way can be defined as a basic type, an array of objects or have also
a certain amount of son. This way to define relations between objects is close to the way
we usually define a grammar in antlr. Some conventions may be respected to produce a
correct tom file, but the description of relations between son and father nodes remind the
rule we use to define a grammar.

Once the specification is written and verified, we can include it into Tom files. Tom
file must be written into host code, but we can include some tom code inside. The best
strength of tom, for our compiler at least, is its ability to match strange patterns into tom
data structures. We can ask for tom to find any node considering its parents, brothers,
uncle, grand-father, son and any other relative family. Contrary to antlr rewrite rule,
we are not locked into a small context of one rule, but we can ask for the matching of a
complete tree, from the root to all the leafs.

This very large pattern matching is the reason that explains the use of tom in this project.
To translate a tree from a language to an other language, especially with two languages
as different as Pluscal2.0 and TLA+, we must change the structure of the tree, and tom
gives us a possibility to choose some very special configuration in the original tree. Fur-
thermore, tom permits to return a new tree (or part of tree) to replace the first one, in
which we can use parts of the old tree. In general, tom gives us a very easy-to-use way
to manipulate tree and by consequent to rewrite trees.

Last but not least, tom comes with a tool already working to interface antlr tree with tom.
The only difficulty of this tool, is to learn to use it and to write patterns that match exactly nodes we are looking for. No more. No less.

2.3 Tlc

Writting a system specification in TLA+ is not very easy for most of the program designers. But it is a very good way to ensure that an algorithm follows some rules, and verifies some postconditions if it’s respect pre-conditions. To do that, we must use a model checker. A model checker is a tool that will create a graph of all possible states of the program environment (variables values). It will also label any state with properties verified by this state. We can then ask for it to ensure that a certain property is always verified, or check that some loop invariants are always true.

Tlc is a model-checker that takes TLA+ input code and uses a configuration file to work. It returns an error if the TLA+ syntax is wrong or if a state doesn’t verify a property it may verify. Tlc was developped to work with lamport’s TLA+. This tool was only used to check that generated code was correct, so it doesn’t take part of the compiler itself. We only paid attention to the way a configuration file must be written. Some Pluscal2.0 codes can indeed be a verification request that we must add to the tlc configuration file.
Chapter 3

Pluscal2.0

After some tests with Sabina’s compiler, Martin QUINSON ask me to start writting a new compiler from the scratch. I took some time to get used to write in TLA+. I read a few pages of ‘Specifying Systems’, the manual lamport wrote as an introduction to TLA+ language. I also was helped by Stephan Mertz, who directed Sabina’s Thesis. Then I started my work following Sabina’s thesis.

3.1 Grammar redactions

First of all, it was necessary to specify the Pluscal2.0 grammar. Jean-Christophe Bach, who helped me using Tom, had written some really basic files, already prepared to work easily with tom and a first draft of Makefile. This permits me to work directly on the grammar without losing too much time trying to have it working with tom. I first wrote an Antlr file to specify the Pluscal2.0 language. The biggest difficulty of this specification were TLA+ expressions in Pluscal2.0 grammar. I defined a TLA+ expression as a group of tokens containing almost all possible tokens. Later in my intership, I defined a second antlr grammar, specifying TLA+ expressions that time. The main objective was to get a lexer/parser, generated by antlr, able to recognize any input programs including TLA+ expressions. The fact of just copying a TLA+ expression would have lead to error raised by TLC (because unnoticed by the Pluscal2.0 compiler). Furthermore, we expected to have a more complete controls on the user utilisation of TLA+ (semantic control, in particular). That was not possible by a simple copy of any TLA+ expressions.

Antlr, hopefully, gives an easy way to use several files linked by including files into each others. The only difficulty was to keep a single Lexer (which means declare all possible tokens all togethers in a single file). Then, rules could be splitted. I therefore specified TLA+ language in a different file than Pluscal2.0. The first try I attempted to specify TLA+, following rigorously Lamport’s definition (‘Specifying systems’ p276) was a failed. The grammar was non-LL. I then use the first compiler Sabina wrote, with Stephan Mertz help, and rewrote the JavaCC version of the grammar in an Antlr file. The grammar was still non-LL, but with some backtracks, antlr was finally able to generate a lexer and a parser that fill our expectations.

To match those two grammars, I had to write two tom specifications. If antlr was able to transform an input stream into a tree, tom was supposed to be used later, for tree transformations. But the way to specify a tom signature was not exactly the same that antlr’s grammar (in other words, no copy-paste). Writting the Pluscal2.0 signature was simple and was done just after writting Pluscal2.0 grammar. For the TLA+ signature, it was written with my first attempt of TLA+ grammar, following strictly lamport’s definition. I did not have to change it too much when using Stephan’s grammar for TLA+.
expressions. It permits to ensure that this new grammar matched the old one.

To finish this first step, I had to complete every antlr files with some rewritting rules describing which tom node must be used to replace an antlr token. The main file was already wrote, thanks to Jean-Christophe.

### 3.2 Tree transformations

From the lexer and the parser, we can obtain a classic antlr tree (object CommonTreeNode). The call of an adaptor permits to get a tom tree. The next part is therefore directly written in java.

First, we wanted to add a semantic control to our tree (it was done at the end of my intership). This control is currently not completed, but permits already to filter and stop the compilation for a lot of common semantic mistakes. The final goal of this semantic control is to force the user to respect his variable declaration like in any programming language. Without this control, we could not keep the control between the Pluscal2.0 and the TLA+ use or acces on variables.

After this control, we must transform a tom tree representing a Pluscal2.0 program into a tom tree representing a TLA+ specification. In memory, there are two different tom specifications (one is used for Pluscal2.0, the other for TLA+). In the end, we will have to transform a Pluscal2.0 node into a TLA+ node. But first, we must definitely change the basic structure of the program.

To do it, we will use an intermediate form, that is still representened in memory by some Pluscal2.0 nodes, but contains a reduce instruction set. Indeed, only 4 instructions are enough to represent a Pluscal2.0 program. Any other statement can be replaced by an equivalent code containing only those four statements. The four statements were choosen to be close to the TLA+ specification. Moving from a Pluscal2.0 way of thinking the program to a TLA+ way, is done by using this intermediate language. Once we have got a simple Pluscal2.0 tree, we will use tom abilities to match any special pattern and rewrite it in a other way. For example, without any close detail on the original language, it is easy to see how a 'loop' of any kind can be transformed using only labels and 'goto' instructions.

For giving a more general impression of the transformation, we will change a succes- sion of action in a succession of state and action. When the Pluscal2.0 language defines a possible situation by using conditions on variables (if a given variable has a given value, for an other variable in a given set... do a certain action) we will create an action (in TLA+, an action is not an affectation or any statement that changes the context, but a formula that explain how to reach a state from an other state) 'guarded' by some conditions. We then will say that the 'action' is equivalent, for exemple, to a given variable having a given value, an other variable belonging to a set, and the next state of a third variable is given by a formula representing the changement, including eventually the actual value of the variable. This is due to the way TLA+ specifies a system. It is given by a conjuction of statements. In those statements, we find conditions on variables to reach this state and actions to apply to reach the next state. This is how, globally, the Pluscal2.0 language, composed of a succession of actions to execute, according to certain conditions, becomes TLA+, composed of actions containing the guard conditions on variables and some instructions to change the context for the next state.

To limit the model checker work, the actions order must be specified. Else, the model
checker will try any possible action in any order, including looping infinitely on the same action. We must also take in consideration the case of a procedure call. To control all those moves, we have to add some instructions to the original program. It is almost equivalent to adding some assembly language instruction to control a PC and a stack context. I will not detail more the main idea that is already explained in Sabina’s thesis. Technically, the work was separated in different steps. First, a normalizer is called, to transform the tree by keeping only the four basic instructions, and an other one that I kept because it simplify a lot the following steps (the goto). Then comes a translator, that both add some instructions about the PC to control the order actions are tried by the model-checker, and translate the four statements in TLA+. Once again, the exact transformation is explained with more detail in Sabina’s thesis.

### 3.3 Code generator

The last step of our compiler is to generate some TLA+ codes. At this point of the program, we obtained a tom tree representing a specification in TLA+. We just have to generate two output files. The first one will contains the TLA+ code. The second was in fact created during the translation. Indeed, when we translated Pluscal2.0 in TLA+, the instruction in Pluscal2.0 that was only written to give to tlc some indications or limitations had no reason to be kept in the TLA+ tree. That’s why the translator returns a tree with only the instructions that must be kept in the TLA+ tree, and it prepares the TLC file at the same time.

For the TLA+ code generation, we use a kind of pretty printer. This printer only generates a code that depends on the tree structure. But it applies the same treatement to any type of node. It’s just a recursive call that walks the tree and returns a string depending on the type of node and the string returned by any son node, if there is some. This simple printer has, therefore no special technical interest.
Chapter 4

Evaluation

All over this project, it was necessary to remain certain that the compiler I was working on was working correctly. The final dreamed objective was to obtain a compiler at least as good as Sabina’s one. But we already knew that a two month internship was not enough to complete the project. Hence, it was necessary to ensure that as much part as possible was working and did not needed any more work.

4.1 Tests

To simplify the project, and make it easier to modify or to enrich, it has been divided in many step. But waiting the end of the project to check the full project could be classified as madness, or a guarantee to spend month trying to debug an unworking code. This is why, instead of a single look to the output of a given input to validate the new code, I stored the outputs in files. Test were kept in some text files, and the compiler generation due to those test was also stored in a text file. Thanks to this way, and with the help of a little script, it was possible to run all the tests, and check at every step, or after any big change in the code that every old test was still validated by the compiler. Even if this meant some work to keep those tests up to date, it permits to avoid some big mistakes. Thanks to test, it was possible to ensure, statement after statement, that each case was correctly treated.

For the compiler, Martin QUINSON suggests me to first complete a transversal work. He suggests me to have a working compiler with only a few statements at the begining. Then I could complete this first draft by adding more and more features to my compiler. If this way of thinking had the advantage to permit some early tests on some specially prepared programs, it also suffers from some complications. Indeed, this way of programming leads to many changes. If we have to change something in the first step of the compilation, all following steps would be impacted. In other word, a single change in the grammar implies to rework on every file of the project. This is what happened when we tried to add a semantic control to the compiler. At this time, the Tom tree was unable to keep any information about the line or column of any tokens. Adding this to the tree (the signature) made me lost a lot of time, repairing every file in the project to have it working with this new signature. This is why, except for this semantic control, I complete some steps before to go further.

Now, grammars and signatures are complete for me. They were completed first, before to start programming anything else. But for the translator, only some statements are currently working. It was possible to limit the number of statements without having many impact on the end of the project. Indeed, regardless of the input tree, the printer had to be the same. The fact that some instructions were ignored did not change anything, and
any change on the translator does not imply a complete rework of the project. Now that my internship is finished, available tests show which instructions are treated.

4.2 Working exemples

To test the translator in general, every step being executed one after the other, I used a simple exemple of algorithm. This algorithm just creates and increases 2 times a variable. The variable then goes by the value 0, 1 and 2. Then I asked for the model checker to ensure that the variables belong to \( \{0,1,2\} \). The model checker raised no error. A second launch with the set 0,1 returned an error and shown an exemple in which the variable was equal to 2. This exemple can be found in the Appendix A. Even if it was very simple, his objective was to check that, in the compiler, every step was working, including the configuration file generation and the tla file generation. It also permits to be sure that tlc accepted the generated code. I spend the end of my internship trying to improve the compiler, but some other exemples based on the same principle could have been created.
Conclusion and future work

At the end of my internship, the compiler is not working as well as Sabina’s one. But its biggest strength is the way it has been programmed. Some parts (grammar and printer) are complete and do not need anything else. The more interesting part, the normalization need to be filled with more statements, but thanks to tom, it will be easy to complete it, like the semantic control. Then, it would be good to add a step to manage the pc value. This is necessary to use different processes or to permit the use of procedure in Pluscal2.0 programs. Currently, procedures are not translated. Concerning processes, the ‘main’ process is the only one that is kept. The translation of other processes in TLA+ is not very difficult. The difficulty is the use of assembly language over TLA+ to ensure that the model checker will follow the actions order and respect the execution of a process.

Once this first compiler will work, it will be possible to add some more instructions. We could then for example give more power on the code atomicity to the user. Or ask for a verification including some preconditions. This could lead to a powerful tool to verify any kind of distributed code, using multiple processes and common variables.

As a conclusion, this internship in the LORIA, gave me a good opportunity to immerse myself in the research environment. It was a great experiment to combine the work of different teams to reach a common objective. The situation of having to reimplement a subject of thesis was perfect for an internship. I was not completely on my own and could rely on the thesis, but Martin QUINSON also gave me the possibility to change the Pluscal2.0 language or to adapt it to my needs. I learned to use Tom, but also TLA+. And I’m glad I had the luck to see a so different way of conceiving a program. That’s why I want to see how the project will evolve in the future.
Thanks

I would like to thanks Martin QUINSON, first because he offered me this internship. But also for his help all over the internship. He was able to create an excellent relationship between trainee and supervisor. But he also gave us a good way to work by forcing us to send him e-mail summarizing our day of work. He also helped us many time before we started to work on a wrong idea. And he took a lot of his own time to have us discovering the research profession, speaking with us and having us to attempt some conferences. For that, thanks you very much Martin!

I also thanks Stephan MERTZ, who directed Sabina thesis. He took some time to help me understanding some TLA+ basics, that saved me a lot of time. But he was also there all over my intership to help me, follow my progression and give me new perspectives about the Pluscal2.0 compiler. Every time we spoke, he gave me a better comprehension of the objective of this compiler.

Finally, I want to thanks Jean-Christophe BACH. He spent a lot of time with me, giving me more and more keys to understand tom. Without his help, I would never have reach this point in the compiler evolution. Thanks for your patience Jean-Christophe!

I can not finish without thanking the whole algorille team, that accept me as a member of the team, without any consideration of the usual trainee/supervisor difference.
Appendix A

A simple compilation

Original Pluscal2.0 code:

(*Header*)

algorithm SimpleClock
extends Naturals
constant N

(*Declaration Section*)

variable a = <<<>, b = 0 , c =0

(*process section*)

(*Main block*)

begin
  lb1 : b :=0;
  lb2 : b :=b+1;
  lb3 : b :=b+1;
end algorithm

invariant b \in 0 .. N

instances N = 3
Generated TLA+ file:

-------------------- MODULE SimpleClock --------------------
EXTENDS Naturals
CONSTANTS N
VARIABLES a, b, c, _pc_Main

vars == 
    <(_pc_Main, a, b, c)>

Init ==
/
\_pc_Main = [element \in {0} |-> "lb1"]
/\a = <<>>
/\b = 0
/\c = 0

lb1 ==
/
\_pc_Main[0] = "lb1"
/
LET
\_b ==
      0
IN
/
\_pc_Main' = [\_pc_Main EXCEPT ![0] = "lb2"]
/\b' = _b
/
UNCHANGED <<a, c>>

lb2 ==
/
\_pc_Main[0] = "lb2"
/
LET
\_b ==
      b + 1
IN
/
\_pc_Main' = [\_pc_Main EXCEPT ![0] = "lb3"]
/\b' = _b
/
UNCHANGED <<a, c>>
lb3 ==
\/
\_/pc>Main[0] == "lb3"
\/
LET
    _b ==
    b + 1
\/
\_/pc>Main’ == [_pc>Main EXCEPT ![0] == "Done"]
\/
_b’ == _b
\/
UNCHANGED <<a, c>>

_Main ==

\/
.lb1
\/
.lb2
\/
.lb3

Next ==

\/
_Main
\/
\/
_/pc>Main[0] == "Done"
\/
UNCHANGED <<vars>>

Spec ==

\/
Init
\/
[ ] [Next]_<<vars>>

_Invariant0 ==
    b \in 0 .. N

============================================= 

Generated cfg file for TLC:

SPECIFICATION Spec
INVARIANT _Invariant0
CONSTANT N = 3