

PGo Manual

For version 0.1.4

February 6, 2019

1 Introduction

TLA+ is a formal specification language, built on the mathematical concepts of set theory and the temporal logic of actions. Using the TLC model checker, TLA+ specifications can be checked exhaustively for specific properties, and the TLA proof system (TLAPS) allows for machine-checked proofs. PlusCal is an algorithm language which can be translated to TLA+ and uses TLA+ as its expression language. It is easy to specify and verify distributed algorithms in PlusCal, thanks to its simple constructs for nondeterminism, concurrency primitives, and rich mathematical constructs. However, PlusCal does not correspond well to a real implementation. The TLA proof system itself does not provide any way of extracting executable code from a PlusCal algorithm, nor does the PlusCal language provide any facilities for describing the kind of interface that extracted code should provide.

PGo aims to correspond a verified PlusCal specification with an executable Go implementation. PGo either compiles a PlusCal algorithm into a corresponding Go implementation with no interface, or accepts a superset of PlusCal called ModularPlusCal that can be compiled into a free-standing Go module that may be used as a library.

Since implementation details like network communication and environmental non-determinism cannot be written in the specification, PGo generates code that is parameterised on implementations of those things. It is then possible to invoke the compiled algorithm using either stock implementations provided by PGo's runtime library or, should the need arise, any implementations that match interface provided.

2 Using PGo

2.1 Installation

Requirements: IntelliJ, Eclipse, or Ant 1.9

- Git clone the source at <https://github.com/UBC-NSS/pgo>
- Option 1: Import as an IntelliJ project
- Option 2: Import as an Eclipse project
- Option 3: Execute `ant build` to compile the project and then execute `pgo.sh [options] pcalfile` to compile `pcalfile`.

Dependencies:

- The Plume options library.
- Java Hamcrest.
- The JSON reference implementation.

PGo was tested on JRE 8, JRE 9, and Go 1.10.

2.2 Execution

To run PGo, run the IntelliJ project, the Eclipse project or run `pgo.sh`. The command-line usage is `pgo [options] pcalfile`.

Optional command line arguments:

```
--version=<boolean>      - Version [default false]
-h --help=<boolean>      - Print usage information [default false]
-q --logLvlQuiet=<boolean> - Reduce printing during execution [default false]
-v --logLvlVerbose=<boolean> - Print detailed information during execution [default false]
-m --mpcalCompile=<boolean> - Compile a Modular PlusCal spec to vanilla PlusCal [default false]
-c --configFilePath=<string> - path to the configuration file, if any
```

2.3 Configuration

PGo requires a JSON configuration file with the following information.

```
1 {
2   "build": {
3     "output_dir": "/path/to/output",
4     "dest_file": "out.go"
5   },
6   "networking": {
7     "enabled": true,
8     "state": {
9       "strategy": "state-server",
10      "endpoints": ["10.0.0.1:1234", "10.0.0.2:1235"],
11      "peers": ["10.0.0.3:4321", "10.0.0.4:4322"],
12      "timeout": 3
13    }
14  },
15  "constants": {
16    "name": "value"
17  }
18 }
```

2.3.1 Build

`output_dir` must point to an existing directory.

`dest_file` specifies the output Go file for PGo to write into. The full path for the file is constructed by appending the value of `dest_file` to `output_dir`. This file will be truncated by PGo.

2.3.2 Networking

`enabled` specifies whether the compiled Go program is a distributed program backed by a network. `enabled` must be `false` when the input PlusCal file is a uniprocess algorithm, otherwise PGo will halt with an error.

`state` specifies the strategy to use for distributed program compilation. It is ignored when `enabled` is `false`. Currently, `etcd` and `state-server` strategies are supported. The default strategy to use is `state-server`.

`peers` specifies a list of peers among which the distributed processes have to establish connections.

`endpoints` specifies the etcd endpoints to which the distributed processes have to connect.

`timeout` specifies the timeout interval in seconds. The default value for this option is 3 seconds.

2.3.3 Constants

The PlusCal algorithm can make use of TLA+ constants that are found outside the algorithm block (i.e. constants declared using the **CONSTANT** keyword). Concrete values for these constants need to be specified in the **constants** dict. Each key is a JSON string containing the name of the constant being defined. Each value is a JSON string containing one valid TLA+ expression.

```
1 {  
2   ...  
3   "constants": {  
4     "myProcs": "{1, 3}",  
5     "N": "3"  
6   },  
7   ...  
8 }
```

Example constant specification

```
1 var myProcs []int  
2 var N int  
3  
4 func init() {  
5     myProcs = []int{1, 3}  
6     N = 3  
7 }
```

Compiled Go

2.4 Type inference

PGo will automatically infer types for variables declared in PlusCal. The type inference algorithm supports a limited form of polymorphism to support different use cases for tuples. Specifically, the tuple literal `<<exp1, exp2, exp3>>` may be compiled as a Go slice literal or a Go struct literal depending on whether `exp1`, `exp2`, and `exp3` have the same type.

2.5 Enabling ModularPlusCal

In order to use ModularPlusCal, there are two changes to the usual PGo workflow:

- In order to model check your ModularPlusCal program, you must first compile it into plain PlusCal in order for it to be interpreted by the TLA+ toolbox. You do this by passing the `-mpcal` command-line option to PGo.
- TODO: how to enable MPCal compilation?

See section 4 for a description of ModularPlusCal features.

2.6 Lock inference

PGo adds locks when compiling a multiprocess PlusCal algorithm. The locking behaviour is described in more detail in 3.9.

3 Translation

PlusCal has many language constructs that are translated to Go in a non-trivial way. These constructs and their translations are described in this section, starting with an overview.

3.1 Overview

3.1.1 PlusCal support

PGo supports both the C and P-syntaxes of PlusCal. Note that unused labels are removed from the Go output and that fresh variable names and labels are generated in order to avoid name capture.

PlusCal feature	Example code	PGo support
Line comment	<code>*line comment</code>	Supported
Block comment	<code>(*block comment *)</code>	Supported
Labelled statements	<pre>label: stmt1; stmt2; * ...</pre>	Compiled with a mutex or a distributed mutex around the statements
While loop	<pre>while (condition) { body; }</pre>	<p>Compiled as</p> <pre>for { if !condition { break } body }</pre>
If statement	<pre>if (condition) { thenPart; } else { elsePart; }</pre>	Supported; compiled as expected
Either statement	<pre>either { stmt1; stmt2; } or { stmt3; stmt4; } or { stmt5; stmt6; } * ...</pre>	<p>Compiled as</p> <pre>case0: stmt1 stmt2 goto endEither case1: stmt3 stmt4 goto endEither case2: stmt5 stmt6 goto endEither // ... endEither:</pre> <p>Each case is tried deterministically from top to bottom (i.e. <code>case0</code> is tried before <code>case1</code>, etc.). Case N is tried only after case 0 to N-1 have failed because await conditions in those cases are not met.</p>
Assignment	<code>x := exp;</code>	Supported; compiled as expected
Multiple variable assignment	<code>x := y y := x + y;</code>	Supported; compiled as multiple assignment in Go
Return statement	<code>return;</code>	Supported; compiled as expected

Skip statement	<code>skip;</code>	Supported; compiled to nothing
Call statement	<code>call procedure1(arg1, arg2);</code>	Supported; compiled as expected
Macro call	<code>macro1(arg1, arg2);</code>	Supported; expanded during compilation
With statement	<code>with (x = exp1, y \in exp2) { body; }</code>	Supported; compiled as variable assignment with fresh names. In the example code, y is assigned the first element of exp2.
Print statement	<code>print exp;</code>	Compiled as <code>fmt.Printf("%v", exp)</code>
Assert statement	<code>assert condition;</code>	Compiled as <code>if !condition { panic("condition"); }</code>
Await statement	<code>await condition;</code>	Compiled as <code>awaitLabel: if !condition { goto awaitLabel }</code>
Goto statement	<code>goto label;</code>	Supported; compiled as expected
Single process algorithm	<code>--algorithm Algo { variables x = exp1, y \in exp2; { body; } }</code>	Supported; compiled as a single-threaded single-process Go program
Multiprocess algorithm	<code>--algorithm Algo { variables x = exp1, y = exp2; process (P \in exp3) variables local = exp4; { body; } }</code>	Supported; compiled with various strategies configured by the user

PlusCal constructs

3.1.2 TLA+ support

Below are the TLA+ constructs. Note that PGo makes liberal use of temporary variables to compile complex TLA+ constructs.

TLA+ feature	Example code	PGo support
Function call	<pre> x[exp1] * or x[exp1, exp2, exp3] * or x[<<exp1, exp2, exp3>>] * or x[[field1 -> e1, field2 -> e2]] </pre>	Supported; compiled code dependent on the type of x (the function)
Binary operator call	<code>x /\ y = z + 1</code>	Supported; compiled as expected
Record	<pre> [field1 \in exp1 -> exp2, field2 -> exp3] </pre>	Unsupported
Function set (as function literal)	<pre> [Nat -> Nat] * or [Nat -> 1..3] </pre>	Unsupported
Function set (as sorted slice of structs)	<code>[1..3 -> 1..3]</code>	Unsupported
Function substitution	<pre> [f EXCEPT ![exp1] = exp2] * or [f EXCEPT !.field = exp] </pre>	Unsupported
If expression	<pre> if condition then thenExp else elseExp </pre>	<p>Compiled as</p> <pre> var result type; if condition { result = thenExp } else { result = elseExp } // result is used in place of // the expression hereafter </pre>
Tuple (as slice)	<code><<exp1, exp2, exp3>></code>	<p>Compiled as slice when all its contents are of the same type</p> <pre> []type{exp1, exp2, exp3} </pre>

Tuple (as struct)	<<exp1, exp2, exp3>>	<p>Compiled as a struct when at least one element is of a different type from the others' types.</p> <pre> struct { e0 type e1 type e2 type }{exp1, exp2, exp3} </pre>
Case expression	<pre> CASE x -> y [] z -> p [] OTHER -> other </pre>	<p>Compiled as</p> <pre> var result type; if x { result = y goto matched } if z { result = p goto matched } result = other matched: // result is used in place of // the expression hereafter </pre>
Existential	<pre> \E a, b, c : exp * or \EE a, b, c : exp </pre>	Unsupportable; TLC chokes when given this expression
Universal	<pre> \A a, b, c : exp * or \AA a, b, c : exp </pre>	Unsupportable; TLC chokes when given this expression
Let expression	<pre> LET op(a, b, c) == exp1 fn[d \in D] == exp2 e == exp3 IN exp </pre>	Unsupported
Assumption	<pre> ASSUME exp * or ASSUMPTION exp * or AXIOM exp </pre>	Unsupported
Theorem	THEOREM exp	Unsupported

Maybe action	[exp1]_exp2	Unsupported
Required action	<<exp1>>_exp2	Unsupported
Operator	Op(arg1, arg2) = exp	Compiled as a Go function
Operator call	Op(exp1, exp2)	Supported; compiled as a function call
Quantified existential	$\exists a \text{ \textit{in} exp1, b \text{ \textit{in} exp2 : exp3}$	<p>Compiled as</p> <pre> exists := false for _, a := range exp1 { for _, b := range exp2 { if exp3 { exists = true goto yes } } } yes: // exists is used in place of // the expression hereafter </pre>
Quantified universal	$\forall a \text{ \textit{in} exp1, b \text{ \textit{in} exp2 : exp3}$	<p>Compiled as</p> <pre> forAll := true for _, a := range exp1 { for _, b := range exp2 { if !exp3 { forAll = false goto no } } } no: // forAll is used in place of // the expression hereafter </pre>
Set constructor	{exp1, exp2, exp3}	<p>Compiled as sorted slice</p> <pre> []type{exp1, exp2, exp3} </pre>
Set comprehension	{exp : a \textit{in} exp1, b \textit{in} exp2}	<p>Compiled as</p> <pre> tmpSet := make([]type, 0) for _, a := range exp1 { for _, b := range exp2 { tmpSet = append(tmpSet, exp) } } // more code to ensure elements in // tmpSet is unique and sorted // tmpSet is used in place of // the expression hereafter </pre>

Set refinement	<code>{a \in exp1 : exp}</code>	Compiled as <pre> tmpSet := make([]type, 0) for _, v := range exp1 { if exp { tmpSet = append(tmpSet, v) } } // tmpSet is used in place of // the expression hereafter </pre>
----------------	---------------------------------	--

TLA+ constructs

3.2 Variable declarations

In addition to the simple variable declaration `var = <val>`, PlusCal supports the declaration `var \in <set>`. This asserts that the initial value of `var` is an element of `<set>`. This is translated into an assignment of the variable `var` to the zeroth element of `<set>`, i.e. `var = tmpSet[0]`.

```

1 variables
2   S = {1, 3};
3   v \in S;
4 {
5   /* algorithm body...
6 }

```

PlusCal

```

1 var S []int
2 var v int
3
4 func init() {
5   S = []int{1, 3}
6   v = S[0]
7 }
8
9 func main() {
10   // algorithm body...
11 }

```

Compiled Go

3.3 Variable assignment

PlusCal supports multiple variable assignment statements: the statement `x := a || y := b` evaluates the right-hand sides first, then assigns the values to the left-hand sides. A common use is swapping the variables `x` and `y` with the statement `x := y || y := x`.

Go has a multiple assignment construct, which fits well as a target for this corresponding PlusCal construct.

```

1 x := y || y := x + y

```

PlusCal

```

1 x, y = y, x+y

```

Compiled Go

3.4 Macros

PlusCal macros have the same semantics as C/C++ `#define` directives. PGo expands the PlusCal macro wherever it occurs.

```

1 variables p = 1, q = 2;
2 macro add(a, b) {
3   a := a + b;
4 }
5 {
6   add(p, q);
7   print p;
8 }

```

PlusCal

```

1 import "fmt"
2
3 var p int = 1
4 var q int = 2
5
6 func main() {
7   p = p + q
8   fmt.Println("%v", p)
9 }

```

Compiled Go

3.5 Data types

PGo supports PlusCal sets, functions, and tuples.

3.5.1 Sets

PlusCal sets are translated into sorted slices in Go.

```

1 variables
2   A = {1, 2, 3};
3   B = {3, 4, 5}
4   C = A \union B;
5 {
6   print A = C;
7 }

```

PlusCal

```

1 A := []int{1, 2, 3}
2 B := []int{3, 4, 5}
3 tmpSet := make([]int, len(A), len(A)+len(B))
4 copy(tmpSet, A)
5 tmpSet = append(tmpSet, B...)
6 sort.Ints(tmpSet)
7 if len(tmpSet) > 1 {
8   previousValue := tmpSet[0]
9   currentIndex := 1
10  for _, v := range tmpSet[1:] {
11    if previousValue != v {
12      tmpSet[currentIndex] = v
13      currentIndex++
14    }
15    previousValue = v
16  }
17  tmpSet = tmpSet[:currentIndex]
18 }
19 C := tmpSet
20 eq := len(A) == len(C)
21 if eq {
22   for i := 0; i < len(A); i++ {
23     eq = A[i] == C[i]
24     if !eq {
25       break
26     }
27   }
28 }
29 fmt.Printf("%v\n", eq)

```

Compiled Go

PlusCal also supports the typical mathematical set constructor notations.

```

1 variables
2   S = {1, 5, 6};
3   T = {2, 3};
4   U = {x \in S : x > 3}; \* equivalent to {5, 6}
5   V = {x + y : x \in S, y \in T}; \* equivalent to {3, 4, 7, 8, 9}
6   \* ...

```

PlusCal

```

1 S := []int{1, 5, 6}
2 T := []int{2, 3}
3 tmpSet := make([]int, 0)
4 for _, x := range S {
5     if x > 3 {
6         tmpSet = append(tmpSet, x)
7     }
8 }
9 U := tmpSet
10 tmpSet0 := make([]int, 0)
11 for _, x := range S {
12     for _, y := range T {
13         tmpSet0 = append(tmpSet0, x+y)
14     }
15 }
16 sort.Ints(tmpSet0)
17 if len(tmpSet0) > 1 {
18     previousValue := tmpSet0[0]
19     currentIndex := 1
20     for _, v := range tmpSet0[1:] {
21         if previousValue != v {
22             tmpSet0[currentIndex] = v
23             currentIndex++
24         }
25         previousValue = v
26     }
27     tmpSet0 = tmpSet0[:currentIndex]
28 }
29 V := tmpSet0
30 // ...

```

Compiled Go

While not as concise as the PlusCal, the output Go code is still readable.

3.5.2 Functions

TLA+ functions with finite domains are translated into sorted slices of structs in Go.

A TLA+ function can be indexed by multiple indices. This is syntactic sugar for a map indexed by a tuple whose components are the indices.

```

1 variables
2   S = {1, 2};
3   f = [x \in S, y \in S |-> x + y];
4   a = f[2, 2]; \* a = 4

```

PlusCal

```

1 S := []int{1, 2}
2 function := make([]struct {
3     key struct {
4         e0 int
5         e1 int
6     }
7     value int
8 }, 0, len(S)*len(S))
9 for _, x := range S {
10     for _, y := range S {
11         function = append(function, struct {
12             key struct {
13                 e0 int
14                 e1 int
15             }
16             value int
17         }{key: struct {
18             e0 int
19             e1 int
20         }{x, y}, value: x + y})
21     }
22 }
23 f := function
24 key := struct {
25     e0 int
26     e1 int
27 }{2, 2}
28 index := sort.Search(len(f), func(i int) bool {
29     return !(f[i].key.e0 < key.e0 || f[i].key.e0 == key.e0 && f[i].key.e1 < key.e1)
30 })
31 a := f[index].value

```

Compiled Go

3.5.3 Tuples

PlusCal tuples are used in several different contexts, so variables involving tuples may have different inferred types depending on their use. Tuples can store homogeneous data, in which case they correspond to Go slices. Tuple components may be of different types, which correspond to Go structs. PlusCal tuples are 1-indexed, but Go tuples and slices are 0-indexed, so 1 is subtracted from all indices in Go.

```

1 variables
2     slice = << "a", "b", "c" >>;
3     tup = << 1, "a" >>;
4 {
5     print slice[2]; \* "b"
6 }

```

PlusCal

```

1 slice := []string{"a", "b", "c"}
2 tup := struct {
3     e0 int
4     e1 string
5 }{1, "a"}
6 fmt.Printf("%v\n", slice[2-1])

```

Compiled Go

3.6 Predicate operations

PlusCal supports the mathematical quantifiers \forall and \exists . PGo compiles these to (nested) for loops, whose bodies check for the relevant condition.

```

1 variables
2   S = {1, 2, 3};
3   T = {4, 5, 6};
4   b1 = \E x \in S : x > 2; \* TRUE
5   b2 = \A x \in S, y \in T : x + y > 6; \* FALSE

```

PlusCal

```

1 S := []int{1, 2, 3}
2 T := []int{4, 5, 6}
3 exists := false
4 for _, x := range S {
5     if x > 2 {
6         exists = true
7         break
8     }
9 }
10 b1 := exists
11 forAll := true
12 for _, x := range S {
13     for _, y := range T {
14         if !(x+y > 6) {
15             forAll = false
16             goto no
17         }
18     }
19 }
20 no:
21 b2 := forAll

```

Compiled Go

3.7 With

The PlusCal `with` statement has the syntax

```

1 variables S_1 = {1, 2, 3}, a = "foo";
2 \* ...
3 with (x_1 \in S_1, x_2 = a, ...)
4 {
5     \* do stuff with x_i ...
6 }

```

PlusCal

This construct selects the first element from each `Si` and assigns them to the local variables `xi`. If the syntax `xi = a` is used, this simply assigns `a` to `xi`. In Go, this translates to

```

1 S_1 := []int{1, 2, 3}
2 a := "foo"
3 x_1 := S_1[0]
4 x_2 := a

```

Compiled Go

The local variables declared by the `with` and its body are potentially renamed to ensure no accidental capture.

3.8 Processes

The PlusCal algorithm body can either contain statements in a uniprocess algorithm, or process declarations in a multiprocess algorithm.

Uniprocess algorithms are translated into single-threaded Go programs.

In a multiprocess algorithm, processes can be declared with the syntax `process (Name \in S)` or `process (Name = Id)`. The first construct spawns a set of processes, one for each ID in the set `S`, and the second spawns a single process with ID `Id`. A process can refer to its identifier with the keyword `self`.

The following is a simple example of a multiprocess PlusCal algorithm, which will be used as the translation source throughout this subsection:

```
1  variables
2      idSet = {1, 2, 3};
3      id = "id";
4
5  process (PName \in idSet)
6  variable local;
7  {
8      local := self;
9  }
10
11 process (Other = id) {
12     print self;
13 }
```

PlusCal

3.8.1 Multi-threaded compilation strategy

With the multi-threaded compilation strategy, PGo converts each process body to a function and spawns a goroutine per process. The function takes a single parameter `self`, the process ID. There are two semantic considerations: the main goroutine should not exit before all goroutines finish executing, and the time at which all child goroutines begin executing should be synchronized. To preserve these semantics, PGo uses a global waitgroup which waits on all goroutines, and each process body pulls from a dummy channel before beginning execution. When all goroutines have been initialized, the dummy channel is closed so that the channel pull no longer blocks, allowing for a synchronized start.

Below is the output Go program when compiled using the multi-threaded compilation strategy. Note that all processes use the same waitgroup (`PGoWait`) and dummy channel (`PGoStart`).

```

1 package main
2
3 import (
4     "fmt"
5     "sync"
6 )
7
8 var idSet []int
9
10 var id string
11
12 var pGoStart chan bool
13
14 var pGoWait sync.WaitGroup
15
16 func init() {
17     idSet = []int{1, 2, 3}
18     id = "id"
19     pGoStart = make(chan bool)
20 }
21
22 func PName(self int) {
23     defer pGoWait.Done()
24     <-pGoStart
25     local := 0
26     local = self
27 }
28
29 func Other(self string) {
30     defer pGoWait.Done()
31     <-pGoStart
32     fmt.Printf("%v\n", self)
33 }
34
35 func main() {
36     for _, v := range idSet {
37         pGoWait.Add(1)
38         go PName(v)
39     }
40     pGoWait.Add(1)
41     go Other(id)
42     close(pGoStart)
43     pGoWait.Wait()
44 }

```

Compiled multi-threaded Go program

3.8.2 Distributed process compilation strategy

TODO

3.9 Labels

In PlusCal, labels are used as targets for `goto` statements and also to specify atomic operations. If a statement is labelled, all statements up to, and excluding, the next label are considered to be a single atomic operation. In Go, unused labels cause compilation errors so PGo only outputs labels when they are targets of some `goto` statement.

To deal with atomicity, PGo divides the global variables into groups and guards each group with a `sync.RWMutex`.

PGo groups variables by performing a set union, merging two variable sets when two variables in them can be accessed in the same label. The following is a simple example:

```
1 variables a = 0, b = 1, c = 2, d = 3;
2 process (P \in {1, 2, 3}) {
3   lab1: a := 1;
4         b := 2;
5   lab2: b := 3;
6         c := 4;
7   lab3: d := 5;
8 }
```

PlusCal

```
1 package main
2
3 import (
4   "sync"
5 )
6
7 var a int
8
9 var b int
10
11 var c int
12
13 var d int
14
15 var pGoStart chan bool
16
17 var pGoWait sync.WaitGroup
18
19 var pGoLock []sync.RWMutex
20
21 func init() {
22     a = 0
23     b = 1
24     c = 2
25     d = 3
26     pGoStart = make(chan bool)
27     pGoLock = make([]sync.RWMutex, 2)
28 }
29
30 func P(self int) {
31     defer pGoWait.Done()
32     <-pGoStart
33     pGoLock[0].Lock()
34     a = 1
35     b = 2
36     pGoLock[0].Unlock()
37     pGoLock[0].Lock()
38     b = 3
39     c = 4
40     pGoLock[0].Unlock()
41     pGoLock[1].Lock()
42     d = 5
43     pGoLock[1].Unlock()
44 }
45
46 func main() {
47     for _, v := range []int{1, 2, 3} {
48         pGoWait.Add(1)
49         go P(v)
50     }
51     close(pGoStart)
52     pGoWait.Wait()
53 }
```

Compiled Go

The variable **b** may be accessed atomically with **a** (in the label **lab1**) and also with **c** (in the label **lab2**) so all three of **a**, **b**, and **c** must be grouped together to prevent data races. PGo locks the correct group before each

atomic operation and unlocks it afterwards. Even single operations must use the lock, since there are no atomicity guarantees for most Go statements. If the atomic operations are specified to be smaller in PlusCal by adding more labels, PGo will compile smaller variable groups, allowing for more parallelism.

3.10 Limitations

Not all PlusCal specifications can be compiled by PGo. This is an overview of some important PlusCal features that are currently unsupported.

- Referencing `self` in a procedure call
- TLA+ features:
 - Alignment of boolean operators in bulleted lists determining precedence
 - Record sets
 - Bags (multisets)
 - The `LET .. IN` construct
 - Temporal logic operators
 - Recursive definitions
 - Builtin modules such as `FiniteSets`.

PGo does not yet have a coherent story for the following desirable features for programmers. However, work on Modular PlusCal which aims to support them is underway.

- Interfacing with other programs
- Reading input from the outside world (e.g. from the command line, from the disk)

4 ModularPlusCal

ModularPlusCal is an extension to PlusCal that aims to add the ability to specify an interface between the actual substance of a PlusCal algorithm and the environment. Modular PlusCal allows the specification writer to more clearly separate abstract and implementation-dependent details, allowing the PGo compiler to generate source code that is easy to change and enables the evolution of specification and implementation to happen at the same time.

4.1 Top-level syntax

Modular PlusCal (MPCal) is comprised of three features: archetypes, mapping macros, and references. MPCal algorithms are declared in `.tla` files as comments as below:

```

1  ---- MODULE DistributedProtocol ----
2  EXTENDS Integers, Sequences, TLC
3
4  CONSTANTS A, B, C
5
6  (*****
7  --mpcal DistributedProtocol {
8    \* Modular PlusCal specification
9  }
10 *****
11 =====

```

MPCal is compiled by PGo to vanilla PlusCal, which is then translated to TLA+ by the TLA+ toolbox. Temporal properties and invariants can then be written as usual.

4.2 Archetypes

A quintessential aspect of modeling the environment is how one should launch a PlusCal algorithm. As is typical of an implementation of a distributed algorithm, someone wanting to deploy it would need to manage things like managing the lifetime and location of the algorithm’s processes, networking, how the processes communicate with each other and how to feed problem-specific data sources into and out of the algorithm, and so forth.

In typical PlusCal, this is effectively impossible. PlusCal only allows the developer to model a specific configuration of an abstract algorithm in order to verify that the algorithm is correct - it does not allow specifying an interface and implementation for the algorithm, such that the “body” of the algorithm may be transformed into an implementation that takes a well-defined set of parameters.

ModularPlusCal’s archetypes address this issue. Instead of specifying a specific process that communicates with its environment via ad-hoc shared global variables, an archetype must communicate with its environment via a set of parameters that it accepts. These are capable of both input and output due to pass by reference parameters, described in section 4.4. You can then either create a model-checking instance of the archetype for model checking or an instance of the compiled archetype in Go, executing it in a real environment.

To declare an archetype, you use a syntax like this:

```
1 archetype YourArchetypeName(ref param1, param2, ...)
2 variable var1, var2 = ...; {
3     labels: statements...
4
5     param1 := var1;
6     var2 := param2;
7 }
```

The syntax is deliberately similar to the syntax for a PlusCal process, the key difference being the parameter list. All PlusCal statements will work as usual in the archetype body, except that access to global variables declared outside the archetype is forbidden, since an archetype should never rely on anything that is not explicitly passed to it as a parameter. Notice that we show the algorithm assigning to `param1`, which is a pass by reference parameter. This is how you send information back into the environment without directly accessing global variables - for further information see the corresponding section of the manual.

Notice also that we do not specify things we would normally specify, like process ID and how many processes there should be. This is because only the instantiator will know these things - most real systems generalise to variable-size structures like single server multiple clients or arbitrary peer to peer. While for model checking purposes it can be useful to assume a particular number of clients or peers, the implementation should not be affected by these assumptions so they are not part of the archetype definition.

Here is a comprehensive list of differences between archetypes and processes:

- Archetypes have more strict scope: they can only access local variables, TLA+ constants, and arguments passed in to them. Access to global variables is not possible; As a consequence, any macros called within an archetype also do not have access to global variables;
- TLA+ operators called within an archetype must both: access no global variables; and be pure.
- Assignments are restricted: only local variables or arguments passed as references can be assigned to (see section 4.4).

Conversely, here is a list of similarities between archetypes and processes

- Same labeling rules apply;
- Archetypes have access to an implicit, immutable self parameter, defined when archetypes are instantiated.

4.2.1 Instantiating for model checking

Since archetypes do not provide much of the information necessary for model checking themselves, we must be able to provide this information separately in order to generate a complete model checking scenario.

We do this via a variation of the `process` declaration called an `instance` declaration. One you have declared your archetype, you can describe how it should be model checked using this syntax:

```
1 process (YourInstanceName = 1) == instance YourArchetypeName(ref param1, param2, ...)
```

This will define a process called `YourInstanceName` with `self=1`, delegating all information about the process body to the archetype `YourArchetypeName`. Just like with PlusCal processes, you can define a set of concurrently executing processes by specifying `YourInstanceName \in someset`. You can also define multiple separate processes as instances of the same archetype if needed.

The parameters `ref param1` and `param2` show the two possible syntaxes for parameter passing. In either case, `param1` and `param2` are required to be already-declared PlusCal global variables.

This instance declaration deliberately matches the syntax example for archetype declaration declaring `YourArchetypeName`, which shows that in order to pass an archetype parameter by reference (see section 4.4) both the parameter declaration and the value passed in must be declared `ref`. This is so that it is clear at both instantiation and declaration that that parameter can be used to mutate the environment.

For a more fleshed out example, consider:

```
1 CONSTANTS COORDINATORS, BACKUPS \* this is declared in the TLA+ code, but is written here for brevity
2
3 variables connection = <<>>,
4         backupConnection = <<>>;
5
6 process (MainCoordinator \in COORDINATORS) == instance Coordinator(connection);
7 process (BackupCoordinator \in BACKUPS) == instance Coordinator(backupConnection);
```

In the definition above, the `connection` variable is global in PlusCal. However, when PGo compiles an specification like the one above, only source code for archetypes is generated. Archetype parameters represent implementation-specific details that need to be filled in by the developer (oftentimes, the PGo runtime will provide most of the logic required in these implementation-specific components).

4.2.2 Instantiating the Go implementation

TODO I don't think we have this down yet

4.3 Mapping macros

Sometimes when writing a PlusCal algorithm it is necessary to consider issues where there is a difference in behaviour between PlusCal variable assignments and the semantics we want to model. For example, while a simple way of representing a network in PlusCal is a shared global variable, this does not model properties like lossy network connections or reordering. Normally in PlusCal the writer will define a set of macros implement the correct modeling behaviour and write the algorithm in terms of those. The problem here is that if you do that then it is in principle impossible to tell apart what the algorithm does and how the environment is modeled.

This cannot be fixed by parameterising the algorithm, since the intended behaviour executes as part of the algorithm (originally via macro expansion or custom TLA+ operators). Instead, we allow the user to specify macros that modify the behaviour of reads and writes to archetype parameters. These are mapping macros.

Mapping macros allow developers to isolate model-checking behavior from archetypes. They are simple wrappers for non-determinism and model checking abstractions.

Suppose we want to model a network that is both lossy and reordering (emulating UDP semantics in concrete environments). MPCal enables the specification developer to write this behavior as a mapping macro:

```

1 mapping macro LossyReorderingNetwork {
2   read {
3     with (msg \in $variable) {
4       $variable := $variable \ msg;
5       yield msg;
6     }
7   }
8
9   write {
10    either { yield $variable } or { yield Append($variable, $value) };
11  }
12 }

```

The mapping macro above introduces a number of related concepts:

- Every mapping macro has a unique identifier: in the previous example, the mapping macro is called `LossyReorderingNetwork`;
- Mapping macros must define two operations: `read` and `write`, which define what happens when the mapped variable is read and written to, respectively. Note that order is relevant: `read` macros must be defined before `write` macros.
- Mapping macros have access to special variables in their definitions: `$variable` is the name of the variable being mapped; `$value` is the value being assigned to the mapped variable.
- `yield` expression indicates that when the mapped variable is read (written to), expression should be read (written) instead.

Mapping macros are supposed to be thin wrappers and, as such, operate under several restrictions:

- Mapping macros cannot reference any variable by name; no variables are in scope.
- `$variable` refers to the name of the variable being mapped and is available on both `read` and `write` mappings; `$value` is the value being written to the mapped variable and therefore is only available in the `write` mapping.
- No labels are allowed; all statements in a mapping macro happen in the same label of the mapped statement (variable read or write).
- Mapping macros cannot create variables whose scope outlives the mapping macro. Locally scoped variables can be created using PlusCal's `with` construct.
- As a corollary of the above, only assignments to `$variable` are permitted, and only on `read` mappings. `Write` mappings cannot write to `$variable` because they are used precisely when an assignment is being made, and PlusCal does not allow writing to the same variable twice in the same step (label).

Once defined, mapping macros can be used during instantiation, mapping variables passed to archetypes:

```

1 process MainCoordinator == instance Coordinator(ref connection)
2   mapping connection via LossyReorderingNetwork;

```

4.3.1 Mapping macros over patterns

Sometimes writes to a single variable are not the things we want to override. Consider a common abstraction for many-to-many networks in PlusCal: a function from process ID to some kind of inbox.

If we pass this to an archetype, the archetype parameter is the entire mapping. It is nearly meaningless to try and map the entire value, since if you were to modify one process's inbox the mapping macro would just see

a new value for the entire mapping, obscuring which inbox changed. This is crucial information, since at the implementation level it makes it impossible to tell the difference between a write to one or many inboxes, turning every network send into some kind of all-to-all broadcast.

Instead, when we apply a mapping macro to an archetype instance declaration we can state that we want to apply the mapping macro to every element of a collection (that is, a TLA+ function) individually.

In the case of our network example, we can write something like this:

```
1 process MainCoordinator == instance Coordinator(ref network)
2   mapping network[_] via LossyReorderingNetwork;
```

Adding `[_]` to a mapping means that whenever the algorithm uses the term `network[...]`, either reading from or assigning to it, the mapping macro is expanded with `network[...]` in its entirety as `$value` rather than just the `network` variable. Since `a.b` is strictly syntax sugar for `a["b"]`, this notation will work on either TLA+ functions or TLA+ records, regardless of whether the indexing or dot notation are used. It is an error to use this notation on a variable that does not belong to either of these types.

Conversely, referring to just `network` on its own no longer has a very useful meaning. If anything, something like assigning to `network` on its own would mean a network-wide broadcast. Since that is rarely intended and complicates things for authors of mapping macros, we have decided to not support this notation in favor of requiring programmers to explicitly write out broadcasts and other aggregate operations if needed. If such an operation is often needed in some algorithm, any such behaviour can easily be encapsulated inside a procedure.

4.4 Pass by reference parameters

References is an extension to parameter passing in PlusCal that makes mutation intent explicit. In particular, they are used when an archetype modifies one of its arguments and also allowing procedures to modify its parameters (not possible in PlusCal).

Assignments to non-local variables in archetypes and procedures can only happen if the argument is passed as a reference:

```
1 procedure inc(ref counter) {
2   i: counter := counter + 1;
3   return;
4 }
5
6 archetype Counter(ref counter) {
7   call inc(ref counter);
8 }
9
10 variable n = 0;
11 process CounterProcess == instance Counter(ref n);
```

In the example above, the keyword `ref` is used to indicate that `n` is passed as a reference to the archetype definition, which is then able to pass it as a reference to the `inc` procedure, which modifies the parameter in a way that is visible after the procedure returns.

5 Example programs

5.1 Euclidean algorithm

The Euclidean algorithm is a simple algorithm that computes the greatest common divisor of two integers, and is a good example PlusCal algorithm.

```
1 ----- MODULE Euclid -----
2 EXTENDS Naturals, TLC
```

```

3  CONSTANT N
4
5  (*
6  --algorithm Euclid {
7    variables u = 24;
8        v \in 1 .. N;
9        v_init = v;
10   {
11   a: while (u # 0) {
12       if (u < v) {
13         u := v || v := u;
14       };
15       b: u := u - v;
16     };
17     print <<24, v_init, "have gcd", v>>
18   }
19 }
20 *)
21 \* BEGIN TRANSLATION
22 VARIABLES u, v, v_init, pc
23
24 vars == << u, v, v_init, pc >>
25
26 Init == (* Global variables *)
27     /\ u = 24
28     /\ v \in 1 .. N
29     /\ v_init = v
30     /\ pc = "a"
31
32 a == /\ pc = "a"
33     /\ IF u # 0
34         THEN /\ IF u < v
35             THEN /\ /\ u' = v
36                 /\ v' = u
37             ELSE /\ TRUE
38                 /\ UNCHANGED << u, v >>
39             /\ pc' = "b"
40         ELSE /\ PrintT(<<24, v_init, "have gcd", v>>)
41             /\ pc' = "Done"
42             /\ UNCHANGED << u, v >>
43     /\ UNCHANGED v_init
44
45 b == /\ pc = "b"
46     /\ u' = u - v
47     /\ pc' = "a"
48     /\ UNCHANGED << v, v_init >>
49
50 Next == a \/ b
51     \/ (* Disjunct to prevent deadlock on termination *)
52     (pc = "Done" /\ UNCHANGED vars)
53
54 Spec == Init /\ [] [Next]_vars
55
56 Termination == <>(pc = "Done")
57
58 \* END TRANSLATION
59 =====

```

PlusCal

```
1 package main
```

```

2
3 import (
4     "fmt"
5 )
6
7 var N int
8
9 func init() {
10     N = 42
11 }
12
13 func main() {
14     u := 24
15     tmpRange := make([]int, N-1+1)
16     for i := 1; i <= N; i++ {
17         tmpRange[i-1] = i
18     }
19     v := tmpRange[0]
20     v_init := v
21     for {
22         if !(u != 0) {
23             break
24         }
25         if u < v {
26             u, v = v, u
27         }
28         u = u - v
29     }
30     fmt.Printf("%v\n", struct {
31         e0 int
32         e1 int
33         e2 string
34         e3 int
35     }{24, v_init, "have gcd", v})
36 }

```

Compiled Go

The constant N needs to be specified in the configuration file whose path is passed to PGo, since its definition does not appear in the comment containing the algorithm. Note the code to swap u and v on line 26 of the Go program.

5.2 N-Queens problem

This PlusCal algorithm computes all possible ways to place n queens on an $n \times n$ chessboard such that no two queens attack each other. It demonstrates the expressive power of PlusCal's set constructs, as the algorithm is very concise.

```

1  ----- MODULE Queens -----
2  EXTENDS Naturals, Sequences, TLC
3  (*****
4  (* Formulation of the N-queens problem and an iterative algorithm to solve *)
5  (* the problem in TLA+. Since there must be exactly one queen in every row *)
6  (* we represent placements of queens as functions of the form *)
7  (* queens \in [ 1..N -> 1..N ] *)
8  (* where queens[i] gives the column of the queen in row i. Note that such *)
9  (* a function is just a sequence of length N. *)
10 (* We will also consider partial solutions, also represented as sequences *)
11 (* of length \leq N. *)
12 (*****
13
14 CONSTANT N \** number of queens and size of the board
15 ASSUME N \in Nat \ {0}

```

```

16
17 (* The following predicate determines if queens i and j attack each other
18    in a placement of queens (represented by a sequence as above). *)
19 Attacks(queens,i,j) ==
20   \/\ queens[i] = queens[j] \** same column
21   \/\ queens[i] - queens[j] = i - j \** first diagonal
22   \/\ queens[j] - queens[i] = i - j \** second diagonal
23
24 (* A placement represents a (partial) solution if no two different queens
25    attack each other in it. *)
26 IsSolution(queens) ==
27   \A i \in 1 .. Len(queens)-1 : \A j \in i+1 .. Len(queens) :
28     ~ Attacks(queens,i,j)
29
30 (* Compute the set of solutions of the N-queens problem. *)
31 Solutions == { queens \in [1..N -> 1..N] : IsSolution(queens) }
32
33 (*****
34 (* We now describe an algorithm that iteratively computes the set of *)
35 (* solutions of the N-queens problem by successively placing queens. *)
36 (* The current state of the algorithm is given by two variables: *)
37 (* - todo contains a set of partial solutions, *)
38 (* - sols contains the set of full solutions found so far. *)
39 (* At every step, the algorithm picks some partial solution and computes *)
40 (* all possible extensions by the next queen. If N queens have been placed *)
41 (* these extensions are in fact full solutions, otherwise they are added *)
42 (* to the set todo. *)
43 (*****
44
45 (* --algorithm QueensPluscal
46    variables
47      todo = { << >> };
48      sols = {};
49
50    begin
51    nxtQ: while todo # {}
52      do
53        with queens \in todo,
54          nxtQ = Len(queens) + 1,
55          cols = { c \in 1..N : ~ \E i \in 1 .. Len(queens) :
56                Attacks( Append(queens, c), i, nxtQ ) },
57          exts = { Append(queens,c) : c \in cols }
58        do
59          if (nxtQ = N)
60            then todo := todo \ {queens}; sols := sols \union exts;
61            else todo := (todo \ {queens}) \union exts;
62          end if;
63        end with;
64      end while;
65      print sols;
66    end algorithm
67 *)
68
69 \** BEGIN TRANSLATION
70 VARIABLES todo, sols, pc
71
72 vars == << todo, sols, pc >>
73
74 Init == (* Global variables *)
75   /\ todo = { << >> }
76   /\ sols = {}
77   /\ pc = "nxtQ"

```



```

78
79 nxtQ == /\ pc = "nxtQ"
80     /\ IF todo # {}
81     THEN /\ \E queens \in todo:
82         LET nxtQ == Len(queens) + 1 IN
83         LET cols == { c \in 1..N : ~ \E i \in 1 .. Len(queens) :
84             Attacks( Append(queens, c), i, nxtQ ) } IN
85         LET exts == { Append(queens,c) : c \in cols } IN
86         IF (nxtQ = N)
87         THEN /\ todo' = todo \ {queens}
88             /\ sols' = (sols \union exts)
89         ELSE /\ todo' = ((todo \ {queens}) \union exts)
90             /\ sols' = sols
91         /\ pc' = "nxtQ"
92     ELSE /\ PrintT(sols)
93         /\ pc' = "Done"
94     /\ UNCHANGED << todo, sols >>
95
96 Next == nxtQ
97     \/ (* Disjunct to prevent deadlock on termination *)
98     (pc = "Done" /\ UNCHANGED vars)
99
100 Spec == Init /\ [] [Next]_vars
101
102 Termination == <>(pc = "Done")
103
104 \** END TRANSLATION
105
106 TypeInvariant ==
107     /\ todo \in SUBSET Seq(1 .. N) /\ \A s \in todo : Len(s) < N
108     /\ sols \in SUBSET Seq(1 .. N) /\ \A s \in sols : Len(s) = N
109
110 (* The set of sols contains only solutions, and contains all solutions
111    when todo is empty. *)
112 Invariant ==
113     /\ sols \subseq Solutions
114     /\ todo = {} => Solutions \subseq sols
115
116 (* Assert that no solutions are ever computed so that TLC displays one *)
117 NoSolutions == sols = {}
118
119 (* Add a fairness condition to ensure progress as long as todo is nonempty *)
120 Liveness == WF_vars(nxtQ)
121 LiveSpec == Spec /\ Liveness
122
123 =====
124 \* Modification History
125 \* Last modified Sat Jun 02 07:28:16 EDT 2018 by osboxes
126 \* Last modified Sat Dec 18 18:57:03 CET 2010 by merz
127 \* Created Sat Dec 11 08:50:24 CET 2010 by merz

```

PlusCal

```

1 package main
2
3 import (
4     "fmt"
5     "sort"
6 )
7
8 var N int

```

```

9
10 func init() {
11     N = 8
12 }
13
14 func Attacks(queens []int, i int, j int) bool {
15     return queens[i-1] == queens[j-1] || queens[i-1]-queens[j-1] == i-j || queens[j-1]-queens[i-1] == i-j
16 }
17
18 func main() {
19     todo := [][]int{[]int{}}
20     sols := [][]int{}
21     for {
22         if !(len(todo) != 0) {
23             break
24         }
25         queens := todo[0]
26         nxtQ := len(queens) + 1
27         tmpSet := make([]int, 0)
28         tmpRange := make([]int, N-1+1)
29         for i := 1; i <= N; i++ {
30             tmpRange[i-1] = i
31         }
32         for _, c := range tmpRange {
33             exists := false
34             tmpRange0 := make([]int, len(queens)-1+1)
35             for i := 1; i <= len(queens); i++ {
36                 tmpRange0[i-1] = i
37             }
38             for _, i := range tmpRange0 {
39                 tmpSlice := make([]int, len(queens), len(queens)+1)
40                 copy(tmpSlice, queens)
41                 tmpSlice = append(tmpSlice, c)
42                 if Attacks(tmpSlice, i, nxtQ) {
43                     exists = true
44                     break
45                 }
46             }
47             if !exists {
48                 tmpSet = append(tmpSet, c)
49             }
50         }
51         cols := tmpSet
52         tmpSet0 := make([][]int, 0)
53         for _, c := range cols {
54             tmpSlice := make([]int, len(queens), len(queens)+1)
55             copy(tmpSlice, queens)
56             tmpSlice = append(tmpSlice, c)
57             tmpSet0 = append(tmpSet0, tmpSlice)
58         }
59         sort.Slice(tmpSet0, func(i int, j int) bool {
60             less := len(tmpSet0[i]) < len(tmpSet0[j])
61             if len(tmpSet0[i]) == len(tmpSet0[j]) {
62                 for i0 := 0; i0 < len(tmpSet0[i]); i0++ {
63                     less = tmpSet0[i][i0] < tmpSet0[j][i0]
64                     if tmpSet0[i][i0] != tmpSet0[j][i0] {
65                         break
66                     }
67                 }
68             }
69             return less
70         })

```

```

71     if len(tmpSet0) > 1 {
72         previousValue := tmpSet0[0]
73         currentIndex := 1
74         for _, v := range tmpSet0[1:] {
75             eq := len(previousValue) == len(v)
76             if eq {
77                 for i0 := 0; i0 < len(previousValue); i0++ {
78                     eq = previousValue[i0] == v[i0]
79                     if !eq {
80                         break
81                     }
82                 }
83             }
84             if !eq {
85                 tmpSet0[currentIndex] = v
86                 currentIndex++
87             }
88             previousValue = v
89         }
90         tmpSet0 = tmpSet0[:currentIndex]
91     }
92     exts := tmpSet0
93     if nxtQ == N {
94         tmpSet1 := make([][]int, 0, len(todo))
95         for _, v := range todo {
96             eq := len(v) == len(queens)
97             if eq {
98                 for i0 := 0; i0 < len(v); i0++ {
99                     eq = v[i0] == queens[i0]
100                     if !eq {
101                         break
102                     }
103                 }
104             }
105             if !eq {
106                 tmpSet1 = append(tmpSet1, v)
107             }
108         }
109         todo = tmpSet1
110         tmpSet2 := make([][]int, len(sols), len(sols)+len(exts))
111         copy(tmpSet2, sols)
112         tmpSet2 = append(tmpSet2, exts...)
113         sort.Slice(tmpSet2, func(i0 int, j0 int) bool {
114             less0 := len(tmpSet2[i0]) < len(tmpSet2[j0])
115             if len(tmpSet2[i0]) == len(tmpSet2[j0]) {
116                 for i1 := 0; i1 < len(tmpSet2[i0]); i1++ {
117                     less0 = tmpSet2[i0][i1] < tmpSet2[j0][i1]
118                     if tmpSet2[i0][i1] != tmpSet2[j0][i1] {
119                         break
120                     }
121                 }
122             }
123             return less0
124         })
125         if len(tmpSet2) > 1 {
126             previousValue := tmpSet2[0]
127             currentIndex := 1
128             for _, v := range tmpSet2[1:] {
129                 eq := len(previousValue) == len(v)
130                 if eq {
131                     for i1 := 0; i1 < len(previousValue); i1++ {
132                         eq = previousValue[i1] == v[i1]

```

```

133         if !eq {
134             break
135         }
136     }
137 }
138 if !eq {
139     tmpSet2[currentIndex] = v
140     currentIndex++
141 }
142 previousValue = v
143 }
144 tmpSet2 = tmpSet2[:currentIndex]
145 }
146 sols = tmpSet2
147 } else {
148     tmpSet1 := make([] []int, 0, len(todo))
149     for _, v := range todo {
150         eq := len(v) == len(queens)
151         if eq {
152             for i0 := 0; i0 < len(v); i0++ {
153                 eq = v[i0] == queens[i0]
154                 if !eq {
155                     break
156                 }
157             }
158         }
159         if !eq {
160             tmpSet1 = append(tmpSet1, v)
161         }
162     }
163     tmpSet2 := make([] []int, len(tmpSet1), len(tmpSet1)+len(ests))
164     copy(tmpSet2, tmpSet1)
165     tmpSet2 = append(tmpSet2, ests...)
166     sort.Slice(tmpSet2, func(i0 int, j0 int) bool {
167         less0 := len(tmpSet2[i0]) < len(tmpSet2[j0])
168         if len(tmpSet2[i0]) == len(tmpSet2[j0]) {
169             for i1 := 0; i1 < len(tmpSet2[i0]); i1++ {
170                 less0 = tmpSet2[i0][i1] < tmpSet2[j0][i1]
171                 if tmpSet2[i0][i1] != tmpSet2[j0][i1] {
172                     break
173                 }
174             }
175         }
176         return less0
177     })
178     if len(tmpSet2) > 1 {
179         previousValue := tmpSet2[0]
180         currentIndex := 1
181         for _, v := range tmpSet2[1:] {
182             eq := len(previousValue) == len(v)
183             if eq {
184                 for i1 := 0; i1 < len(previousValue); i1++ {
185                     eq = previousValue[i1] == v[i1]
186                     if !eq {
187                         break
188                     }
189                 }
190             }
191             if !eq {
192                 tmpSet2[currentIndex] = v
193                 currentIndex++
194             }

```

```

195         previousValue = v
196     }
197     tmpSet2 = tmpSet2[:currentIndex]
198 }
199     todo = tmpSet2
200 }
201 }
202 fmt.Printf("%v\n", sols)
203 }

```

Compiled Go

Note that the non-trivial types for all variables are correctly inferred by PGo.

5.3 Dijkstra's mutex algorithm

This is a multiprocess algorithm which only allows one process to be in the critical section at one time.

```

1  ----- MODULE DijkstraMutex -----
2  (*****
3  (* This is a PlusCal version of the first published mutual exclusion *)
4  (* algorithm, which appeared in *)
5  (* *)
6  (* E. W. Dijkstra: "Solution of a Problem in Concurrent *)
7  (* Programming Control". Communications of the ACM 8, 9 *)
8  (* (September 1965) page 569. *)
9  (* *)
10 (* Here is the description of the algorithm as it appeared in that paper. *)
11 (* The global variables are declared by *)
12 (* *)
13 (* Boolean array b, c[1:N]; integer k *)
14 (* *)
15 (* The initial values of b[i] and c[i] are true, for each i in 1..N. The *)
16 (* initial value of k can be any integer in 1..N. The pseudo-code for the *)
17 (* i-th process, for each i in 1..N, is: *)
18 (* *)
19 (* integer j; *)
20 (* Li0: b[i] := false; *)
21 (* Li1: if k # i then *)
22 (* Li2: begin c[i] := true; *)
23 (* Li3: if b[k] then k := i; *)
24 (* go to Li1 *)
25 (* end *)
26 (* else *)
27 (* Li4: begin c[i] := false; *)
28 (* for j := 1 step 1 until N do *)
29 (* if j # i and not c[j] then go to Li1 *)
30 (* end; *)
31 (* critical section; *)
32 (* c[i] := true; b[i] := true; *)
33 (* remainder of the cycle in which stopping is allowed; *)
34 (* go to Li0 *)
35 (* *)
36 (* It appears to me that the "else" preceding label Li4 begins the else *)
37 (* clause of the if statement beginning at Li1, and that the code from Li4 *)
38 (* through the end three lines later should be the body of that else *)
39 (* clause. However, the indentation indicates otherwise. Moreover, that *)
40 (* interpretation produces an incorrect algorithm. It seems that this *)
41 (* "else" actually marks an empty else clause for the if statement at Li1. *)
42 (* (Perhaps there should have been a semicolon after the "else".) *)
43 (*****
44

```

```

45 EXTENDS Integers
46
47 (*****
48 (* There is no reason why the processes need to be numbered from 1 to N. *)
49 (* So, we assume an arbitrary set Proc of process names. *)
50 (*****
51 CONSTANT Proc
52
53 (*****
54 Here is the PlusCal version of this algorithm.
55 The algorithm was modified from the original by adding a the variable temp2,
56 to avoid a type consistency conflict when temp changes type at Li4a.
57
58 --algorithm Mutex
59 { variables b = [i \in Proc |-> TRUE], c = [i \in Proc |-> TRUE], k \in Proc;
60   process (P \in Proc)
61     variable temp, temp2 ;
62     { Li0: while (TRUE)
63       { b[self] := FALSE;
64         Li1: if (k # self) { Li2: c[self] := TRUE;
65           Li3a: temp := k;
66           Li3b: if (b[temp]) { Li3c: k := self } ;
67           Li3d: goto Li1
68         };
69         Li4a: c[self] := FALSE;
70         temp2 := Proc \ {self};
71         Li4b: while (temp2 # {})
72           { with (j \in temp2)
73             { temp2 := temp2 \ {j};
74               if (~c[j]) { goto Li1 }
75             }
76           };
77         cs: skip; /* the critical section
78         Li5: c[self] := TRUE;
79         Li6: b[self] := TRUE;
80         ncs: skip /* non-critical section ("remainder of cycle")
81       }
82     }
83 }
84 Notes on the PlusCal version:
85
86 1. Label Li3d is required by the translation. It could be eliminated by
87 adding a then clause to the if statement of Li3b and putting the goto
88 in both branches of the if statement.
89
90 2. The for loop in section Li4 of the original has been changed to
91 a while loop that examines the other processes in an arbitrary
92 (nondeterministically chosen) order. Because temp is set equal
93 to the set of all processes other than self, there is no need for
94 a test corresponding to the "if j # i" in the original. Note that
95 the process-local variable j has been replaced by the identifier
96 j that is local to the with statement.
97 *****
98 /* BEGIN TRANSLATION
99 CONSTANT defaultInitValue
100 VARIABLES b, c, k, pc, temp
101
102 vars == << b, c, k, pc, temp >>
103
104 ProcSet == (Proc)
105
106 Init == (* Global variables *)

```

```

107 /\ b = [i \in Proc |-> TRUE]
108 /\ c = [i \in Proc |-> TRUE]
109 /\ k \in Proc
110 (* Process P *)
111 /\ temp = [self \in Proc |-> defaultInitValue]
112 /\ pc = [self \in ProcSet |-> CASE self \in Proc -> "Li0"]
113
114 Li0(self) == /\ pc[self] = "Li0"
115 /\ b' = [b EXCEPT ![self] = FALSE]
116 /\ pc' = [pc EXCEPT ![self] = "Li1"]
117 /\ UNCHANGED << c, k, temp >>
118
119 Li1(self) == /\ pc[self] = "Li1"
120 /\ IF k # self
121 THEN /\ pc' = [pc EXCEPT ![self] = "Li2"]
122 ELSE /\ pc' = [pc EXCEPT ![self] = "Li4a"]
123 /\ UNCHANGED << b, c, k, temp >>
124
125 Li2(self) == /\ pc[self] = "Li2"
126 /\ c' = [c EXCEPT ![self] = TRUE]
127 /\ pc' = [pc EXCEPT ![self] = "Li3a"]
128 /\ UNCHANGED << b, k, temp >>
129
130 Li3a(self) == /\ pc[self] = "Li3a"
131 /\ temp' = [temp EXCEPT ![self] = k]
132 /\ pc' = [pc EXCEPT ![self] = "Li3b"]
133 /\ UNCHANGED << b, c, k >>
134
135 Li3b(self) == /\ pc[self] = "Li3b"
136 /\ IF b[temp[self]]
137 THEN /\ pc' = [pc EXCEPT ![self] = "Li3c"]
138 ELSE /\ pc' = [pc EXCEPT ![self] = "Li3d"]
139 /\ UNCHANGED << b, c, k, temp >>
140
141 Li3c(self) == /\ pc[self] = "Li3c"
142 /\ k' = self
143 /\ pc' = [pc EXCEPT ![self] = "Li3d"]
144 /\ UNCHANGED << b, c, temp >>
145
146 Li3d(self) == /\ pc[self] = "Li3d"
147 /\ pc' = [pc EXCEPT ![self] = "Li1"]
148 /\ UNCHANGED << b, c, k, temp >>
149
150 Li4a(self) == /\ pc[self] = "Li4a"
151 /\ c' = [c EXCEPT ![self] = FALSE]
152 /\ temp' = [temp EXCEPT ![self] = Proc \ {self}]
153 /\ pc' = [pc EXCEPT ![self] = "Li4b"]
154 /\ UNCHANGED << b, k >>
155
156 Li4b(self) == /\ pc[self] = "Li4b"
157 /\ IF temp[self] # {}
158 THEN /\ \E j \in temp[self]:
159 /\ temp' = [temp EXCEPT
160 /\ [self] = temp[self] \ {j}]
161 /\ IF ~c[j]
162 THEN /\ pc' = [pc EXCEPT
163 /\ [self] = "Li1"]
164 ELSE /\ pc' = [pc EXCEPT
165 /\ [self] = "Li4b"]
166 ELSE /\ pc' = [pc EXCEPT ![self] = "cs"]
167 /\ UNCHANGED temp
168 /\ UNCHANGED << b, c, k >>

```

```

169
170 cs(self) == /\ pc[self] = "cs"
171         /\ TRUE
172         /\ pc' = [pc EXCEPT ![self] = "Li5"]
173         /\ UNCHANGED << b, c, k, temp >>
174
175 Li5(self) == /\ pc[self] = "Li5"
176         /\ c' = [c EXCEPT ![self] = TRUE]
177         /\ pc' = [pc EXCEPT ![self] = "Li6"]
178         /\ UNCHANGED << b, k, temp >>
179
180 Li6(self) == /\ pc[self] = "Li6"
181         /\ b' = [b EXCEPT ![self] = TRUE]
182         /\ pc' = [pc EXCEPT ![self] = "ncs"]
183         /\ UNCHANGED << c, k, temp >>
184
185 ncs(self) == /\ pc[self] = "ncs"
186         /\ TRUE
187         /\ pc' = [pc EXCEPT ![self] = "Li0"]
188         /\ UNCHANGED << b, c, k, temp >>
189
190 P(self) == Li0(self) \/ Li1(self) \/ Li2(self) \/ Li3a(self) \/ Li3b(self)
191         \/ Li3c(self) \/ Li3d(self) \/ Li4a(self) \/ Li4b(self)
192         \/ cs(self) \/ Li5(self) \/ Li6(self) \/ ncs(self)
193
194 Next == (\E self \in Proc: P(self))
195         \/ (* Disjunct to prevent deadlock on termination *)
196         ((\A self \in ProcSet: pc[self] = "Done") /\ UNCHANGED vars)
197
198 Spec == Init /\ [] [Next]_vars /\ \A self \in Proc: WF_vars(P(self))
199
200 Termination == <>(\A self \in ProcSet: pc[self] = "Done")
201
202 \* END TRANSLATION
203
204 (*****
205 (* The following formula asserts that no two processes are in their *)
206 (* critical sections at the same time. It is the invariant that a mutual *)
207 (* exclusion algorithm should satisfy. You can have TLC check that the *)
208 (* algorithm is a mutual exclusion algorithm by checking that this formula *)
209 (* is an invariant. *)
210 (*****
211 MutualExclusion == \A i, j \in Proc :
212         (i # j) => ~ /\ pc[i] = "cs"
213         /\ pc[j] = "cs"
214 (*****
215 (* An equivalent way to perform the same test would be to change the *)
216 (* statement labeled cs (the critical section) to *)
217 (* *)
218 (* cs: assert \A j \in Proc \ {self} : pc[j] # "cs" *)
219 (* *)
220 (* You can give this a try. However, the assert statement requires that *)
221 (* the EXTENDS statement also import the standard module TLC, so it should *)
222 (* read *)
223 (* *)
224 (* EXTENDS Integers, TLC *)
225 (*****
226
227 (*****
228 (* LIVENESS *)
229 (* *)
230 (* If you are a sophisticated PlusCal user and know a little temporal *)

```



```

231 (* logic, you can continue reading about the liveness properties of the *)
232 (* algorithm. *)
233 (* *)
234 (* Dijkstra's algorithm is "deadlock free", which for a mutual exclusion *)
235 (* algorithm means that if some process is trying to enter its critical *)
236 (* section, then some process (not necessarily the same one) will *)
237 (* eventually enter its critical section. Since a process begins trying *)
238 (* to enter its critical section when it is at the control point labeled *)
239 (* Li0, and it is in its critical section when it is at control point cs, *)
240 (* the following formula asserts deadlock freedom. *)
241 (*****
242 DeadlockFree == \A i \in Proc :
243     (pc[i] = "Li0") ~> (\E j \in Proc : pc[j] = "cs")
244 (*****
245 (* Dijkstra's algorithm is deadlock free only under the assumption of *)
246 (* fairness of process execution. The simplest such fairness assumption *)
247 (* is weak fairness on each process's next-state action. This means that *)
248 (* no process can halt if it is always possible for that process to take a *)
249 (* step. The following statement tells the PlusCal translator to define *)
250 (* the specification to assert weak fairness of each process's next-state *)
251 (* action. *)
252 (* *)
253 (* PlusCal options (wf) *)
254 (* *)
255 (* This statement can occur anywhere in the file--either in a comment or *)
256 (* before or after the module. Because of this statement, the translator *)
257 (* has added the necessary fairness conjunct to the definition of Spec. *)
258 (* So, you can have the TLC model checker check that the algorithm *)
259 (* satisfies property DeadlockFree. *)
260 (*****
261
262 (*****
263 (* Dijkstra's algorithm is not "starvation free", because it allows some *)
264 (* waiting processes to "starve", never entering their critical section *)
265 (* while other processes keep entering and leaving their critical *)
266 (* sections. Starvation freedom is asserted by the following formula. *)
267 (* You can use TLC to show that the algorithm is not starvation free by *)
268 (* producing a counterexample trace. *)
269 (*****
270 StarvationFree == \A i \in Proc :
271     (pc[i] = "Li0") ~> (pc[i] = "cs")
272
273 (*****
274 (* In this algorithm, no process can ever be blocked waiting at an 'await' *)
275 (* statement or a 'with (v \in S)' statement with S empty. Therefore, *)
276 (* weak fairness of each process means that each process keeps continually *)
277 (* trying to enter its critical section, and it exits the critical *)
278 (* section. An important requirement of a mutual exclusion solution, one *)
279 (* that rules out many simple solutions, is that a process is allowed to *)
280 (* remain forever in its non-critical section. (There is also no need to *)
281 (* require that a process that enters its critical section ever leaves it, *)
282 (* though without that requirement the definition of starvation freedom *)
283 (* must be changed.) *)
284 (* *)
285 (* We can allow a process to remain forever in its critical section by *)
286 (* replacing the 'skip' statement that represents the non-critical section *)
287 (* with the following statement, which allows the process to loop forever. *)
288 (* *)
289 (* ncs: either skip or goto ncs *)
290 (* *)
291 (* An equivalent non-critical section is *)
292 (* *)

```

```

293 (* nsc: either skip or await FALSE *)
294 (* *)
295 (* A more elegant method is to change the fairness requirement to assert *)
296 (* weak fairness of a process's next-state action only when the process is *)
297 (* not in its non-critical section. This is accomplished by taking the *)
298 (* following formula LSpec as the algorithm's specification. *)
299 (*****)
300 LSpec == Init /\ [] [Next]_vars
301         /\ \A self \in Proc: WF_vars((pc[self] # "nsc") /\ P(self))
302
303 (*****)
304 (* If we allow a process to remain forever in its non-critical section, *)
305 (* then our definition of deadlock freedom is too weak. Suppose process p *)
306 (* were in its critical section and process q, trying to enter its *)
307 (* critical section, reached Li1. Formula DeadlockFree would allow a *)
308 (* behavior in which process q exited its critical section and remained *)
309 (* forever in its non-critical section, but process p looped forever *)
310 (* trying to enter its critical section and never succeeding. To rule out *)
311 (* this possibility, we must replace the formula *)
312 (* *)
313 (* pc[i] = "Li0" *)
314 (* *)
315 (* in DeadLock free with one asserting that control in process i is *)
316 (* anywhere in control points Li0 through Li4b. It's easier to express *)
317 (* this by saying where control in process i is NOT, which we do in the *)
318 (* following property. *)
319 (*****)
320 DeadlockFreedom ==
321     \A i \in Proc :
322     (pc[i] \notin {"Li5", "Li6", "nsc"}) ~> (\E j \in Proc : pc[j] = "cs")
323 (*****)
324 (* Do you see why it's not necessary to include "cs" in the set of values *)
325 (* that pc[i] does not equal? *)
326 (*****)
327
328
329
330 (*****)
331 (* Using a single worker thread on a 2.5GHz dual-processor computer, TLC *)
332 (* can check MutualExclusion and liveness of a 3-process model in about 2 *)
333 (* or 3 minutes (depending on which spec is used and which liveness *)
334 (* property is checked). That model has 90882 reachable states and a *)
335 (* state graph of diameter 54. TLC can check a 4-process model in about *)
336 (* 53 minutes. That model has 33288512 reachable states and a state graph *)
337 (* of diameter 89. *)
338 (*****)
339 =====
340 \* Modification History
341 \* Last modified Sun Dec 31 22:04:29 EST 2017 by osboxes
342 \* Last modified Sat Jan 01 12:14:14 PST 2011 by lamport
343 \* Created Fri Dec 31 14:14:14 PST 2010 by lamport

```

PlusCal

```

1 package main
2
3 import (
4     "sort"
5     "sync"
6 )
7

```

```

8  var Proc []int
9
10 var b []struct {
11     key int
12     value bool
13 }
14
15 var c []struct {
16     key int
17     value bool
18 }
19
20 var k int
21
22 var pGoStart chan bool
23
24 var pGoWait sync.WaitGroup
25
26 var pGoLock []sync.RWMutex
27
28 func init() {
29     tmpRange := make([]int, 3-1+1)
30     for i := 1; i <= 3; i++ {
31         tmpRange[i-1] = i
32     }
33     Proc = tmpRange
34     function := make([]struct {
35         key int
36         value bool
37     }, 0, len(Proc))
38     for _, i := range Proc {
39         function = append(function, struct {
40             key int
41             value bool
42         }{key: i, value: true})
43     }
44     b = function
45     function0 := make([]struct {
46         key int
47         value bool
48     }, 0, len(Proc))
49     for _, i := range Proc {
50         function0 = append(function0, struct {
51             key int
52             value bool
53         }{key: i, value: true})
54     }
55     c = function0
56     k = Proc[0]
57     pGoStart = make(chan bool)
58     pGoLock = make([]sync.RWMutex, 3)
59 }
60
61 func P(self int) {
62     defer pGoWait.Done()
63     <-pGoStart
64     temp := 0
65     temp2 := []int{}
66     pGoLock[2].Lock()
67     for {
68         if !true {
69             pGoLock[2].Unlock()

```

```

70         break
71     }
72     key := self
73     index := sort.Search(len(b), func(i int) bool {
74         return !(b[i].key < key)
75     })
76     b[index].value = false
77     pGoLock[2].Unlock()
78 Li1:
79     pGoLock[1].Lock()
80     if k != self {
81         pGoLock[1].Unlock()
82         pGoLock[0].Lock()
83         key0 := self
84         index0 := sort.Search(len(c), func(i0 int) bool {
85             return !(c[i0].key < key0)
86         })
87         c[index0].value = true
88         pGoLock[0].Unlock()
89         pGoLock[1].Lock()
90         temp = k
91         pGoLock[1].Unlock()
92         pGoLock[2].Lock()
93         key1 := temp
94         index1 := sort.Search(len(b), func(i1 int) bool {
95             return !(b[i1].key < key1)
96         })
97         if b[index1].value {
98             pGoLock[2].Unlock()
99             pGoLock[1].Lock()
100             k = self
101             pGoLock[1].Unlock()
102         } else {
103             pGoLock[2].Unlock()
104         }
105         goto Li1
106     } else {
107         pGoLock[1].Unlock()
108     }
109     pGoLock[0].Lock()
110     key0 := self
111     index0 := sort.Search(len(c), func(i0 int) bool {
112         return !(c[i0].key < key0)
113     })
114     c[index0].value = false
115     tmpSet := make([]int, 0, len(Proc))
116     for _, v := range Proc {
117         if v != self {
118             tmpSet = append(tmpSet, v)
119         }
120     }
121     temp2 = tmpSet
122     pGoLock[0].Unlock()
123     pGoLock[0].Lock()
124     for {
125         if !(len(temp2) != 0) {
126             break
127         }
128         j := temp2[0]
129         tmpSet0 := make([]int, 0, len(temp2))
130         for _, v := range temp2 {
131             if v != j {

```

```

132         tmpSet0 = append(tmpSet0, v)
133     }
134 }
135 temp2 = tmpSet0
136 key1 := j
137 index1 := sort.Search(len(c), func(i1 int) bool {
138     return !(c[i1].key < key1)
139 })
140 if !c[index1].value {
141     pGoLock[0].Unlock()
142     goto Li1
143 }
144 }
145 pGoLock[0].Unlock()
146 pGoLock[0].Lock()
147 key1 := self
148 index1 := sort.Search(len(c), func(i1 int) bool {
149     return !(c[i1].key < key1)
150 })
151 c[index1].value = true
152 pGoLock[0].Unlock()
153 pGoLock[2].Lock()
154 key2 := self
155 index2 := sort.Search(len(b), func(i2 int) bool {
156     return !(b[i2].key < key2)
157 })
158 b[index2].value = true
159 pGoLock[2].Unlock()
160 pGoLock[2].Lock()
161 }
162 pGoLock[2].Unlock()
163 }
164
165 func main() {
166     for _, v := range Proc {
167         pGoWait.Add(1)
168         go P(v)
169     }
170     close(pGoStart)
171     pGoWait.Wait()
172 }

```

Compiled Go

The constant Proc is defined to be 1 .. 3 in the configuration. If the process set needs to be changed, only the configuration needs to be edited.

5.4 Load balancer

This is a Modular PlusCal multiprocess algorithm that implements a load balancing system. The processes include the load balancer, the servers, and the clients.

```

1  ----- MODULE load_balancer -----
2  (*****
3  (* Specifies a simple load balancer. *)
4  (*****
5
6  \* Extends some built-in TLA+ modules
7  EXTENDS Naturals, Sequences, TLC
8
9  \* The 'TCPChannel' mapping macro used in this specification is parameterized
10 \* by a 'BUFFER_SIZE' constant. This value controls the number of messages being

```

```

11  \* held in a buffer by each process. Processes trying to send a message to another
12  \* process with a full buffer wil "block" (not be scheduled by TLC).
13  CONSTANT BUFFER_SIZE
14
15  \* Define a constant identifier for the load balancer.
16  LoadBalancerId == 0
17
18  \* The number of servers and clients in the model checking setup.
19  CONSTANTS NUM_SERVERS, NUM_CLIENTS
20
21  \* TLC should assume that both numbers are greater than zero (i.e., we always
22  \* have at least one server and one client). Note, however, that increasing
23  \* these numbers makes the number of states to be checked by TLC to grow
24  \* exponentially.
25  ASSUME NUM_SERVERS > 0 /\ NUM_CLIENTS > 0
26
27  \* GET_PAGE is a label attached to messages sent from the clients to
28  \* the load balancer.
29  \*
30  \* WEB_PAGE abstractly represents a web page that the server may return
31  \* to clients. The content of the webpage is, obviously, orthogonal to the
32  \* correctness of our load balancer.
33  CONSTANTS GET_PAGE, WEB_PAGE
34
35  \* total nodes in the system:
36  \* number of clients + number of servers + the load balancer
37  NUM_NODES == NUM_CLIENTS + NUM_SERVERS + 1
38
39  (*****
40  --mpcal LoadBalancer {
41
42    \* The TCPChannel mapping macro models a communication channel
43    \* between two processes using TCP-like semantics. In particular:
44    \*
45    \* - reading from the channel "blocks" unless there is a message
46    \* ready to be read.
47    \* - message delivery is reliable and ordered (i.e., FIFO).
48    mapping macro TCPChannel {
49      read {
50        await Len($variable) > 0;
51        with (msg = Head($variable)) {
52          $variable := Tail($variable);
53          yield msg;
54        };
55      }
56
57      write {
58        await Len($variable) < BUFFER_SIZE;
59        yield Append($variable, $value);
60      }
61    }
62
63    \* Mapping macros keep implementation-specific behavior that we don't
64    \* want to model check outside of our archetype definitions.
65    \* In the case of this load balancer, how a server retrieves a web page
66    \* is orthogonal to the correctness of the properties we are interested
67    \* to check with this specification.
68    \*
69    \* This mapping macro abstracts the process of reading a web page by
70    \* always returning the 'WEB_PAGE' constant when the variable is read.
71    \*
72    \* Since "writing" to this mapping is meaningless, the attempting to write

```

```

73  \* to a variable mapped with WebPage will result in a model checking
74  \* error (see 'assert(FALSE)' in the write definition).
75  mapping macro WebPages {
76      read {
77          yield WEB_PAGE;
78      }
79
80      write {
81          assert(FALSE);
82          yield $value;
83      }
84  }
85
86  \* ALoadBalancer is the archetype that defines the behavior of
87  \* the load balancer process. The 'mailboxes' parameter represents
88  \* connections to all nodes in the system.
89  archetype ALoadBalancer(ref mailboxes)
90
91  \* Local variables of this archetype:
92  variables
93      \* Holds messages received by the load balancer (sent
94      \* by clients)
95      msg,
96
97      \* identifier attached to every message sent to servers by
98      \* the load balancer.
99      next = 0;
100 {
101     main:
102     while (TRUE) {
103
104         \* waits for a message to be received. Upon receipt, the 'assert'
105         \* call ensures that the message is of type 'GET_PAGE', the only
106         \* type of message supported in this simple specification.
107         \*
108         \* Every message received by the load balancer is expected to
109         \* be a tuple in the format:
110         \*
111         \* << {message_type}, {client_id} >>
112         \*
113         \* Note that tuples are 1-indexed.
114         rcvMsg:
115             msg := mailboxes[LoadBalancerId];
116             assert(msg[1] = GET_PAGE);
117
118         \* the load balancer needs to forward the client request to the
119         \* server, who will process the request and send a web page back to
120         \* the client.
121         \*
122         \* The message sent to the server is a tuple in the format:
123         \*
124         \* << {message_id}, {client_id} >>
125         \*
126         \* We send the client ID here so that the server can directly
127         \* reply to a client, bypassing the load balancer. This is usually
128         \* not what happens in practice, but the model is simple
129         \* enough for our (illustrative) purposes.
130         sendServer:
131             next := (next % NUM_SERVERS) + 1;
132             mailboxes[next] := << next, msg[2] >>;
133     }
134 }

```

```

135
136  \* AServer is the archetype that defines the behavior of the servers
137  \* in our system. The two parameters it receives are:
138  \*
139  \* - mailboxes: contains connections to every node in the system
140  \* - page_stream: a source of web pages for the server. In practice,
141  \* this is implementation specific and irrelevant for
142  \* the properties we want to check in this specification
143  archetype AServer(ref mailboxes, page_stream)
144
145  \* Local variables
146  variable
147      \* temporary buffer to hold incoming messages
148      msg;
149  {
150      serverLoop:
151      while (TRUE) {
152
153          \* waits for an incoming message. Note that the only process
154          \* that sends messages to the server is the load balancer process
155          \* (defined according to the ALoadBalancer archetype) and the
156          \* message has the format << {message_id}, {client_id} >>
157          rcvReq:
158              msg := mailboxes[self];
159
160          sendPage:
161              \* sends a web page (read from 'page_stream') back to the requester
162              \* i.e., the client.
163              mailboxes[msg[2]] := page_stream;
164      }
165  }
166
167  \* GLOBAL VARIABLES *\
168
169  variables
170      \* our network is modeled as a function from node identifier
171      \* to a sequence of incoming messages
172      network = [id \in 0..NUM_NODES |-> <<>>],
173
174      \* the stream of web pages to be served by the server. Since we
175      \* intend to map this variable using the WebPages mapping macro,
176      \* the initial value assigned to it here is irrelevant.
177      stream = 0;
178
179  \* PROCESS INSTANTIATION *\
180
181  \* The system has a single load balancer entity, instantiated from the ALoadBalancer
182  \* archetype. The model of our network is going to be the one defined by the TCPChannel
183  \* mapping macro in all instantiations.
184  fair process (LoadBalancer = LoadBalancerId) == instance ALoadBalancer(ref network)
185      mapping network[_] via TCPChannel;
186
187  \* Instantiate 'NUM_SERVERS' server processes according to the AServer archetype.
188  \* We map the page stream according to the WebPages mapping macro since this is
189  \* an implementation detail that needs to be specified during implementation at
190  \* a later stage.
191  fair process (Servers \in 1..NUM_SERVERS) == instance AServer(ref network, stream)
192      mapping network[_] via TCPChannel
193      mapping stream via WebPages;
194
195  \* We have a load balancer that waits for messages and servers that are ready
196  \* to serve web pages when the load balancer requests. However, there is one

```



```

197  \* piece missing from this: a _client_ that actually drives the other two
198  \* components.
199  \*
200  \* To illustrate how regular PlusCal processes can be used with Modular PlusCal,
201  \* we model the client as a vanilla PlusCal process.
202
203
204
205  \* Client processes are given integer identifiers starting from NUM_SERVERS+1.
206  \* Keep in mind that this "range" notation in PlusCal defines a set, and is
207  \* _inclusive_ (i.e., NUM_SERVERS+NUM_CLIENTS+1 is part of the set).
208  \*
209  \* Also note that since the client needs to send network messages to processes
210  \* defined by the previous archetypes, we need to have the exact same network
211  \* model here. However, since mapping macros are not available for regular
212  \* PlusCal processes, we need to copy the specification of the TCPChannel
213  \* mapping macro in this client definition.
214  fair process (Client \in (NUM_SERVERS+1)..(NUM_SERVERS+NUM_CLIENTS+1))
215
216  \* Local variables
217  variable
218      \* Temporary buffer to hold incoming messages
219      msg;
220  {
221      clientLoop:
222          while (TRUE) {
223
224              \* First, the client makes a request to the load balancer.
225              \* The format of the message is a tuple: << {message_type}, {client_id} >>.
226              \* If you check the ALoadBalancer definition, this is the message format
227              \* expected there.
228              \*
229              \* Remember that 'self' is an implicitly defined, immutable variable that
230              \* contains the process identifier of the "running" process.
231              clientRequest:
232                  msg := << GET_PAGE, self >>;
233                  await Len(network[LoadBalancerId]) < BUFFER_SIZE;
234                  network[LoadBalancerId] := Append(network[LoadBalancerId], msg);
235
236
237              \* Clients then wait for the response to the previously sent request.
238              \* Since there is only one type of web page in this simple specification
239              \* (defined by the WEB_PAGE constant), we assert here that the message
240              \* received indeed is equal our expected web page.
241              clientReceive:
242                  await Len(network[self]) > 0;
243                  with (m = Head(network[self])) {
244                      network[self] := Tail(network[self]);
245                      assert(m = WEB_PAGE);
246                  };
247          }
248  }
249
250
251  \* BEGIN PLUSCAL TRANSLATION
252  --algorithm LoadBalancer {
253      variables network = [id \in (0)..(NUM_NODES) |-> <<>>], stream = 0;
254      fair process (LoadBalancer = LoadBalancerId)
255      variables msg, next = 0, mailboxesRead, mailboxesWrite, mailboxesWrite0;
256      {
257          main:
258              if (TRUE) {

```

```

259         rcvMsg:
260             await (Len(network[LoadBalancerId]))>(0);
261             with (msg0 = Head(network[LoadBalancerId])) {
262                 mailboxesWrite := [network EXCEPT ![LoadBalancerId] = Tail(network[LoadBalancerId])];
263                 mailboxesRead := msg0;
264             };
265             msg := mailboxesRead;
266             assert (msg[1])=(GET_PAGE);
267             network := mailboxesWrite;
268
269         sendServer:
270             next := ((next)%(NUM_SERVERS))+1;
271             await (Len(network[next]))<(BUFFER_SIZE);
272             mailboxesWrite := [network EXCEPT ![next] = Append(network[next], <<next, msg[2]>>)];
273             network := mailboxesWrite;
274             goto main;
275
276     } else {
277         mailboxesWrite0 := network;
278         network := mailboxesWrite0;
279     };
280
281 }
282 fair process (Servers \in (1)..(NUM_SERVERS))
283 variables msg, mailboxesRead0, mailboxesWrite1, page_streamRead, mailboxesWrite2;
284 {
285     serverLoop:
286         if (TRUE) {
287             rcvReq:
288                 await (Len(network[self]))>(0);
289                 with (msg1 = Head(network[self])) {
290                     mailboxesWrite1 := [network EXCEPT ![self] = Tail(network[self])];
291                     mailboxesRead0 := msg1;
292                 };
293                 msg := mailboxesRead0;
294                 network := mailboxesWrite1;
295
296             sendPage:
297                 page_streamRead := WEB_PAGE;
298                 await (Len(network[msg[2]]))<(BUFFER_SIZE);
299                 mailboxesWrite1 := [network EXCEPT ![msg[2]] = Append(network[msg[2]], page_streamRead)];
300                 network := mailboxesWrite1;
301                 goto serverLoop;
302
303         } else {
304             mailboxesWrite2 := network;
305             network := mailboxesWrite2;
306         };
307
308 }
309 fair process (Client \in ((NUM_SERVERS)+(1))..(((NUM_SERVERS)+(NUM_CLIENTS))+1))
310 variables msg;
311 {
312     clientLoop:
313         while (TRUE) {
314             clientRequest:
315                 msg := <<GET_PAGE, self>>;
316                 await (Len(network[LoadBalancerId]))<(BUFFER_SIZE);
317                 network[LoadBalancerId] := Append(network[LoadBalancerId], msg);
318
319             clientReceive:
320                 await (Len(network[self]))>(0);

```

```

321         with (m = Head(network[self])) {
322             network[self] := Tail(network[self]);
323             assert (m)=(WEB_PAGE);
324         };
325
326     };
327
328 }
329 }
330 /* END PLUSCAL TRANSLATION
331
332
333
334 *****
335 /* BEGIN TRANSLATION
336 /* Process variable msg of process LoadBalancer at line 255 col 15 changed to msg_
337 /* Process variable msg of process Servers at line 283 col 15 changed to msg_S
338 CONSTANT defaultInitValue
339 VARIABLES network, stream, pc, msg_, next, mailboxesRead, mailboxesWrite,
340             mailboxesWrite0, msg_S, mailboxesRead0, mailboxesWrite1,
341             page_streamRead, mailboxesWrite2, msg
342
343 vars == << network, stream, pc, msg_, next, mailboxesRead, mailboxesWrite,
344             mailboxesWrite0, msg_S, mailboxesRead0, mailboxesWrite1,
345             page_streamRead, mailboxesWrite2, msg >>
346
347 ProcSet == {LoadBalancerId} \cup ((1)..(NUM_SERVERS)) \cup (((NUM_SERVERS)+(1))..((NUM_SERVERS)+(NUM_CLIENTS))+(1)))
348
349 Init == (* Global variables *)
350         /\ network = [id \in (0)..(NUM_NODES) |-> <<>>]
351         /\ stream = 0
352         (* Process LoadBalancer *)
353         /\ msg_ = defaultInitValue
354         /\ next = 0
355         /\ mailboxesRead = defaultInitValue
356         /\ mailboxesWrite = defaultInitValue
357         /\ mailboxesWrite0 = defaultInitValue
358         (* Process Servers *)
359         /\ msg_S = [self \in (1)..(NUM_SERVERS) |-> defaultInitValue]
360         /\ mailboxesRead0 = [self \in (1)..(NUM_SERVERS) |-> defaultInitValue]
361         /\ mailboxesWrite1 = [self \in (1)..(NUM_SERVERS) |-> defaultInitValue]
362         /\ page_streamRead = [self \in (1)..(NUM_SERVERS) |-> defaultInitValue]
363         /\ mailboxesWrite2 = [self \in (1)..(NUM_SERVERS) |-> defaultInitValue]
364         (* Process Client *)
365         /\ msg = [self \in ((NUM_SERVERS)+(1))..((NUM_SERVERS)+(NUM_CLIENTS))+(1)) |-> defaultInitValue]
366         /\ pc = [self \in ProcSet |-> CASE self = LoadBalancerId -> "main"
367                 [] self \in (1)..(NUM_SERVERS) -> "serverLoop"
368                 [] self \in ((NUM_SERVERS)+(1))..((NUM_SERVERS)+(NUM_CLIENTS))+(1)) -> "clientLoop"]
369
370 main == /\ pc[LoadBalancerId] = "main"
371         /\ IF TRUE
372             THEN /\ pc' = [pc EXCEPT ![LoadBalancerId] = "rcvMsg"]
373                 /\ UNCHANGED << network, mailboxesWrite0 >>
374             ELSE /\ mailboxesWrite0' = network
375                 /\ network' = mailboxesWrite0'
376                 /\ pc' = [pc EXCEPT ![LoadBalancerId] = "Done"]
377         /\ UNCHANGED << stream, msg_, next, mailboxesRead, mailboxesWrite,
378                 msg_S, mailboxesRead0, mailboxesWrite1,
379                 page_streamRead, mailboxesWrite2, msg >>
380
381 rcvMsg == /\ pc[LoadBalancerId] = "rcvMsg"
382           /\ (Len(network[LoadBalancerId]))>(0)

```

```

383      /\ LET msg0 == Head(network[LoadBalancerId]) IN
384      /\ mailboxesWrite' = [network EXCEPT ![LoadBalancerId] = Tail(network[LoadBalancerId])]
385      /\ mailboxesRead' = msg0
386      /\ msg_ = mailboxesRead'
387      /\ Assert((msg_[1])=(GET_PAGE),
388      "Failure of assertion at line 266, column 21.")
389      /\ network' = mailboxesWrite'
390      /\ pc' = [pc EXCEPT ![LoadBalancerId] = "sendServer"]
391      /\ UNCHANGED << stream, next, mailboxesWrite0, msg_S, mailboxesRead0,
392      mailboxesWrite1, page_streamRead, mailboxesWrite2,
393      msg >>
394
395 sendServer == /\ pc[LoadBalancerId] = "sendServer"
396      /\ next' = ((next)%(NUM_SERVERS))+1
397      /\ (Len(network[next']))<(BUFFER_SIZE)
398      /\ mailboxesWrite' = [network EXCEPT ![next'] = Append(network[next'], <<next', msg_[2]>>)]
399      /\ network' = mailboxesWrite'
400      /\ pc' = [pc EXCEPT ![LoadBalancerId] = "main"]
401      /\ UNCHANGED << stream, msg_, mailboxesRead, mailboxesWrite0,
402      msg_S, mailboxesRead0, mailboxesWrite1,
403      page_streamRead, mailboxesWrite2, msg >>
404
405 LoadBalancer == main \/ rcvMsg \/ sendServer
406
407 serverLoop(self) == /\ pc[self] = "serverLoop"
408      /\ IF TRUE
409      THEN /\ pc' = [pc EXCEPT ![self] = "rcvReq"]
410      /\ UNCHANGED << network, mailboxesWrite2 >>
411      ELSE /\ mailboxesWrite2' = [mailboxesWrite2 EXCEPT ![self] = network]
412      /\ network' = mailboxesWrite2'[self]
413      /\ pc' = [pc EXCEPT ![self] = "Done"]
414      /\ UNCHANGED << stream, msg_, next, mailboxesRead,
415      mailboxesWrite, mailboxesWrite0, msg_S,
416      mailboxesRead0, mailboxesWrite1,
417      page_streamRead, msg >>
418
419 rcvReq(self) == /\ pc[self] = "rcvReq"
420      /\ (Len(network[self]))>(0)
421      /\ LET msg1 == Head(network[self]) IN
422      /\ mailboxesWrite1' = [mailboxesWrite1 EXCEPT ![self] = [network EXCEPT ![self] = Tail(network[self]
423      /\ mailboxesRead0' = [mailboxesRead0 EXCEPT ![self] = msg1]
424      /\ msg_S' = [msg_S EXCEPT ![self] = mailboxesRead0'[self]]
425      /\ network' = mailboxesWrite1'[self]
426      /\ pc' = [pc EXCEPT ![self] = "sendPage"]
427      /\ UNCHANGED << stream, msg_, next, mailboxesRead,
428      mailboxesWrite, mailboxesWrite0,
429      page_streamRead, mailboxesWrite2, msg >>
430
431 sendPage(self) == /\ pc[self] = "sendPage"
432      /\ page_streamRead' = [page_streamRead EXCEPT ![self] = WEB_PAGE]
433      /\ (Len(network[msg_S[self][2]]))<(BUFFER_SIZE)
434      /\ mailboxesWrite1' = [mailboxesWrite1 EXCEPT ![self] = [network EXCEPT ![msg_S[self][2]] = Append(net
435      /\ network' = mailboxesWrite1'[self]
436      /\ pc' = [pc EXCEPT ![self] = "serverLoop"]
437      /\ UNCHANGED << stream, msg_, next, mailboxesRead,
438      mailboxesWrite, mailboxesWrite0, msg_S,
439      mailboxesRead0, mailboxesWrite2, msg >>
440
441 Servers(self) == serverLoop(self) \/ rcvReq(self) \/ sendPage(self)
442
443 clientLoop(self) == /\ pc[self] = "clientLoop"
444      /\ pc' = [pc EXCEPT ![self] = "clientRequest"]

```

```

445         /\ UNCHANGED << network, stream, msg_, next, mailboxesRead,
446             mailboxesWrite, mailboxesWrite0, msg_S,
447             mailboxesRead0, mailboxesWrite1,
448             page_streamRead, mailboxesWrite2, msg >>
449
450 clientRequest(self) == /\ pc[self] = "clientRequest"
451     /\ msg' = [msg EXCEPT ![self] = <<GET_PAGE, self>>]
452     /\ (Len(network[LoadBalancerId]))<(BUFFER_SIZE)
453     /\ network' = [network EXCEPT ![LoadBalancerId] = Append(network[LoadBalancerId], msg'[self])]
454     /\ pc' = [pc EXCEPT ![self] = "clientReceive"]
455     /\ UNCHANGED << stream, msg_, next, mailboxesRead,
456         mailboxesWrite, mailboxesWrite0, msg_S,
457         mailboxesRead0, mailboxesWrite1,
458         page_streamRead, mailboxesWrite2 >>
459
460 clientReceive(self) == /\ pc[self] = "clientReceive"
461     /\ (Len(network[self]))>(0)
462     /\ LET m == Head(network[self]) IN
463         /\ network' = [network EXCEPT ![self] = Tail(network[self])]
464         /\ Assert((m)=(WEB_PAGE),
465             "Failure of assertion at line 323, column 25.")
466     /\ pc' = [pc EXCEPT ![self] = "clientLoop"]
467     /\ UNCHANGED << stream, msg_, next, mailboxesRead,
468         mailboxesWrite, mailboxesWrite0, msg_S,
469         mailboxesRead0, mailboxesWrite1,
470         page_streamRead, mailboxesWrite2, msg >>
471
472 Client(self) == clientLoop(self) \/ clientRequest(self)
473     \/ clientReceive(self)
474
475 Next == LoadBalancer
476     \/ (\E self \in (1)..(NUM_SERVERS): Servers(self))
477     \/ (\E self \in ((NUM_SERVERS)+1)..((NUM_SERVERS)+(NUM_CLIENTS))+1): Client(self))
478
479 Spec == /\ Init /\ [][Next]_vars
480     /\ WF_vars(LoadBalancer)
481     /\ \A self \in (1)..(NUM_SERVERS) : WF_vars(Servers(self))
482     /\ \A self \in ((NUM_SERVERS)+1)..((NUM_SERVERS)+(NUM_CLIENTS))+1 : WF_vars(Client(self))
483
484 \* END TRANSLATION
485
486
487 (* INVARIANTS *)
488
489
490 \* This is an _invariant_ of our specification: in other words,
491 \* we expect the BuffersOk predicate to always be true in every step of execution
492 BufferOk(node) == Len(network[node]) >= 0 /\ Len(network[node]) <= BUFFER_SIZE
493 BuffersOk == \A node \in DOMAIN network : BufferOk(node)
494
495
496 (* PROPERTIES *)
497
498 \* This is a property we would like to check about our specification.
499 \* Properties are defined using _temporal logic_. In this specific example,
500 \* we want to make sure that every client that requests a web page (i.e., are
501 \* in the 'clientRequest' label) eventually receive a response (i.e., are
502 \* in the 'clientReceive' label). In order to specify this property, we have to
503 \* write the formula as if the client enters 'clientReceive' label, it will
504 \* eventually successfully receive a response and then go back to issuing
505 \* another request in the 'clientRequest' label.
506 ReceivesPage(client) == pc[client] = "clientReceive" ~> pc[client] = "clientRequest"

```

```
507 ClientsOk == \A client \in (NUM_SERVERS+1)..(NUM_SERVERS+NUM_CLIENTS+1) : ReceivesPage(client)
508
509 =====
510 \* Modification History
511 \* Last modified Wed Feb 06 17:26:45 PST 2019 by minh
512 \* Last modified Sat Feb 02 19:59:51 PST 2019 by rmc
```

Modular PlusCal