PGo Manual

For version 0.1.4

February 7, 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.

```
{
    "build": {
        "output_dir": "/path/to/output",
        "dest_file": "out.go"
    },
    "networking": {
        "enabled": true,
        "state": {
            "strategy": "state-server",
            "endpoints": ["10.0.0.1:1234", "10.0.0.2:1235"],
            "peers": ["10.0.0.3:4321", "10.0.0.4:4322"],
            "timeout": 3
        }
    },
    "constants": {
        "name": "value"
    }
}
```

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.

```
{
  ...
  "constants": {
    "myProcs": "{1, 3}",
    "N": "3"
  },
  ...
}
```

Example constant specification

```go
var myProcs []int
var N int

func init() {
    myProcs = []int{1, 3}
    N = 3
}
```

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 `-m` command-line option to PGo.
- TODO: how to enable MPCal to Go 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	`\*line comment`	Supported
Block comment	`(*block comment *)`	Supported

Labelled statements	`label:`
	`  stmt1;`
	`  stmt2;`
	`  \* ...`
		Compiled with a mutex or a distributed mutex around the statements

While loop	`while (condition) {`
	`  body;`
	`}`
		Compiled as
		`for {`
		`  if !condition {`
		`    break`
		`  }`
		`  body`
		`}`

If statement	`if (condition) {`
	`  thenPart;`
	`} else {`
	`  elsePart;`
	`}`
		Supported; compiled as expected

Either statement	`either { stmt1; stmt2; }`
	`    or { stmt3; stmt4; }`
	`    or { stmt5; stmt6; }`
	`    \* ...`
		Compiled as
		`case0:`
		`  stmt1`
		`  stmt2`
		`  goto endEither`
		`case1:`
		`  stmt3`
		`  stmt4`
		`  goto endEither`
		`case2:`
		`  stmt5`
		`  stmt6`
		`  goto endEither`
		`// ...`
		`endEither:`
		Each case is tried deterministically from top to bottom (i.e. `case0` is tried before `case1`, etc.). Case N is tried only after case 0 to N-1 have failed because await conditions in those cases are not met.

Assignment	`x := exp;`	Supported; compiled as expected
Multiple variable assignment	`x := y || y := x + y;`	Supported; compiled as multiple assignment in Go
Return statement	`return;`	Supported; compiled as expected

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

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.

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

Tuple (as struct)	`<<exp1, exp2, exp3>>`	Compiled as a struct when at least one element is of a different type from the others' types.
		`struct {`
		`  e0 type`
		`  e1 type`
		`  e2 type`
		`}{exp1, exp2, exp3}`

Case expression	`CASE x -> y`
	`  [] z -> p`
	`  [] OTHER -> other`
		Compiled as
		`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`

Existental	`\E a, b, c : exp`
	`\* or`
	`\EE a, b, c : exp`
		Unsupportable; TLC chokes when given this expression

Universal	`\A a, b, c : exp`
	`\* or`
	`\AA a, b, c : exp`
		Unsupportable; TLC chokes when given this expression

Let expression	`LET op(a, b, c) == exp1`
	`    fn[d \in D] == exp2`
	`    e == exp3`
	`IN exp`
		Unsupported

Assumption	`ASSUME exp`
	`\* or`
	`ASSUMPTION exp`
	`\* or`
	`AXIOM exp`
		Unsupported

Theorem	`THEOREM exp`	Unsupported

<table>
<tr><td>Maybe action</td><td><code>[exp1]_exp2</code></td><td>Unsupported</td></tr>
<tr><td>Required action</td><td><code><<exp1>>_exp2</code></td><td>Unsupported</td></tr>
<tr><td>Operator</td><td><code>Op(arg1, arg2) = exp</code></td><td>Compiled as a Go function</td></tr>
<tr><td>Operator call</td><td><code>Op(exp1, exp2)</code></td><td>Supported; compiled as a function call</td></tr>
<tr><td>Quantified existential</td><td><code>\E a \in exp1, b \in exp2 : exp3</code></td><td>Compiled as
<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></td></tr>
<tr><td>Quantified universal</td><td><code>\A a \in exp1, b \in exp2 : exp3</code></td><td>Compiled as
<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></td></tr>
<tr><td>Set constructor</td><td><code>{exp1, exp2, exp3}</code></td><td>Compiled as sorted slice
<pre>
[]type{exp1, exp2, exp3}
</pre></td></tr>
<tr><td>Set comprehension</td><td><code>{exp : a \in exp1, b \in exp2}</code></td><td>Compiled as
<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></td></tr>
</table>

<table>
<tr><td>Set refinement</td><td><code>{a \in exp1 : exp}</code></td><td>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></td></tr>
</table>

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]`.

```
variables
    S = {1, 3};
    v \in S;
{
    \* algorithm body...
}
```

PlusCal

```
var S []int
var v int

func init() {
    S = []int{1, 3}
    v = S[0]
}

func main() {
    // algorithm body...
}
```

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.

```
x := y || y := x + y
```

PlusCal

```
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.

```
variables p = 1, q = 2;
macro add(a, b) {
    a := a + b;
}
{
    add(p, q);
    print p;
}
```

PlusCal

```
import "fmt"

var p int = 1
var q int = 2

func main() {
    p = p + q
    fmt.Println("%v", p)
}
```

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.

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

PlusCal

```
A := []int{1, 2, 3}
B := []int{3, 4, 5}
tmpSet := make([]int, len(A), len(A)+len(B))
copy(tmpSet, A)
tmpSet = append(tmpSet, B...)
sort.Ints(tmpSet)
if len(tmpSet) > 1 {
    previousValue := tmpSet[0]
    currentIndex := 1
    for _, v := range tmpSet[1:] {
        if previousValue != v {
            tmpSet[currentIndex] = v
            currentIndex++
        }
        previousValue = v
    }
    tmpSet = tmpSet[:currentIndex]
}
C := tmpSet
eq := len(A) == len(C)
if eq {
    for i := 0; i < len(A); i++ {
        eq = A[i] == C[i]
        if !eq {
            break
        }
    }
}
fmt.Printf("%v\n", eq)
```

Compiled Go

PlusCal also supports the typical mathematical set constructor notations.

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

PlusCal

```
S := []int{1, 5, 6}
T := []int{2, 3}
tmpSet := make([]int, 0)
for _, x := range S {
    if x > 3 {
        tmpSet = append(tmpSet, x)
    }
}
U := tmpSet
tmpSet0 := make([]int, 0)
for _, x := range S {
    for _, y := range T {
        tmpSet0 = append(tmpSet0, x+y)
    }
}
sort.Ints(tmpSet0)
if len(tmpSet0) > 1 {
    previousValue := tmpSet0[0]
    currentIndex := 1
    for _, v := range tmpSet0[1:] {
        if previousValue != v {
            tmpSet0[currentIndex] = v
            currentIndex++
        }
        previousValue = v
    }
    tmpSet0 = tmpSet0[:currentIndex]
}
V := tmpSet0
// ...
```

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.

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

PlusCal

```go
S := []int{1, 2}
function := make([]struct {
    key struct {
        e0 int
        e1 int
    }
    value int
}, 0, len(S)*len(S))
for _, x := range S {
    for _, y := range S {
        function = append(function, struct {
            key struct {
                e0 int
                e1 int
            }
            value int
        }{key: struct {
            e0 int
            e1 int
        }{x, y}, value: x + y})
    }
}
f := function
key := struct {
    e0 int
    e1 int
}{2, 2}
index := sort.Search(len(f), func(i int) bool {
    return !(f[i].key.e0 < key.e0 || f[i].key.e0 == key.e0 && f[i].key.e1 < key.e1)
})
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.

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

PlusCal

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

Compiled Go

3.6 Predicate operations

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

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

PlusCal

```
S := []int{1, 2, 3}
T := []int{4, 5, 6}
exists := false
for _, x := range S {
    if x > 2 {
        exists = true
        break
    }
}
b1 := exists
forAll := true
for _, x := range S {
    for _, y := range T {
        if !(x+y > 6) {
            forAll = false
            goto no
        }
    }
}
no:
b2 := forAll
```

Compiled Go

3.7 With

The PlusCal `with` statement has the syntax

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

PlusCal

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

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

```
variables
    idSet = {1, 2, 3};
    id = "id";

process (PName \in idSet)
variable local;
{
    local := self;
}

process (Other = id) {
    print self;
}
```

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`).

```go
package main

import (
	"fmt"
	"sync"
)

var idSet []int

var id string

var pGoStart chan bool

var pGoWait sync.WaitGroup

func init() {
	idSet = []int{1, 2, 3}
	id = "id"
	pGoStart = make(chan bool)
}

func PName(self int) {
	defer pGoWait.Done()
	<-pGoStart
	local := 0
	local = self
}

func Other(self string) {
	defer pGoWait.Done()
	<-pGoStart
	fmt.Printf("%v\n", self)
}

func main() {
	for _, v := range idSet {
		pGoWait.Add(1)
		go PName(v)
	}
	pGoWait.Add(1)
	go Other(id)
	close(pGoStart)
	pGoWait.Wait()
}
```

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:

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

PlusCal

```go
package main

import (
"sync"
)

var a int

var b int

var c int

var d int

var pGoStart chan bool

var pGoWait sync.WaitGroup

var pGoLock []sync.RWMutex

func init() {
    a = 0
    b = 1
    c = 2
    d = 3
    pGoStart = make(chan bool)
    pGoLock = make([]sync.RWMutex, 2)
}

func P(self int) {
    defer pGoWait.Done()
    <-pGoStart
    pGoLock[0].Lock()
    a = 1
    b = 2
    pGoLock[0].Unlock()
    pGoLock[0].Lock()
    b = 3
    c = 4
    pGoLock[0].Unlock()
    pGoLock[1].Lock()
    d = 5
    pGoLock[1].Unlock()
}

func main() {
    for _, v := range []int{1, 2, 3} {
        pGoWait.Add(1)
        go P(v)
    }
    close(pGoStart)
    pGoWait.Wait()
}
```

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:

```
---- MODULE DistributedProtocol ----
EXTENDS Integers, Sequeneces, TLC

CONSTANTS A, B, C

(***********************
--mpcal DistributedProtocol {
   \* Modular PlusCal specification
}
***********************)
====================================
```

MPCal is compiled by PGo to vanilla PlusCal, which is turn 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 speficic 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:

```
archetype YourArchetypeName(ref param1, param2, ...)
variable var1, var2 = ...; {
    labels: statements...

    param1 := var1;
    var2 := param2;
}
```

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:

```
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 declatation 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:

```
CONSTANTS COORDINATORS, BACKUPS \* this is declared in the TLA+ code, but is written here for brevity

variables connection = <<>>,
          backupConnection = <<>>;

process (MainCoordinator \in COORDINATORS) == instance Coordinator(connection);
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:

```
mapping macro LossyReorderingNetwork {
    read {
        with (msg \in $variable) {
            $variable := $variable \ msg;
            yield msg;
        }
    }

    write {
        either { yield $variable } or { yield Append($variable, $value) };
    }
}
```

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: `$variable` is the name of the variable being mapped; `$variable` 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.
- `$variable` refers to the name of the variable being mapped and is available on both read and write mappings; `$variable` 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:

```
process MainCoordinator == instance Coordinator(ref connection)
    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:

```
process MainCoordinator == instance Coordinator(ref network)
    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:

```
procedure inc(ref counter) {
    i: counter := counter + 1;
    return;
}

archetype Counter(ref counter) {
    call inc(ref counter);
}

variable n = 0;
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.

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

```
CONSTANT N

(*
--algorithm Euclid {
  variables u = 24;
            v \in 1 .. N;
            v_init = v;
  {
  a: while (u # 0) {
      if (u < v) {
          u := v || v := u;
      };
  b: u := u - v;
    };
    print <<24, v_init, "have gcd", v>>
  }
}
*)
\* BEGIN TRANSLATION
VARIABLES u, v, v_init, pc

vars == << u, v, v_init, pc >>

Init == (* Global variables *)
        /\ u = 24
        /\ v \in 1 .. N
        /\ v_init = v
        /\ pc = "a"

a == /\ pc = "a"
     /\ IF u # 0
           THEN /\ IF u < v
                      THEN /\ /\ u' = v
                              /\ v' = u
                      ELSE /\ TRUE
                           /\ UNCHANGED << u, v >>
                /\ pc' = "b"
           ELSE /\ PrintT(<<24, v_init, "have gcd", v>>)
                /\ pc' = "Done"
                /\ UNCHANGED << u, v >>
     /\ UNCHANGED v_init

b == /\ pc = "b"
     /\ u' = u - v
     /\ pc' = "a"
     /\ UNCHANGED << v, v_init >>

Next == a \/ b
           \/ (* Disjunct to prevent deadlock on termination *)
              (pc = "Done" /\ UNCHANGED vars)

Spec == Init /\ [][Next]_vars

Termination == <>(pc = "Done")

\* END TRANSLATION
===================================================================
```

PlusCal

```
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.

```
-------------------------- MODULE Queens -----------------------------
EXTENDS Naturals, Sequences, TLC
(***************************************************************************)
(* Formulation of the N-queens problem and an iterative algorithm to solve *)
(* the problem in TLA+. Since there must be exactly one queen in every row *)
(* we represent placements of queens as functions of the form *)
(* queens \in [ 1..N -> 1..N ] *)
(* where queens[i] gives the column of the queen in row i. Note that such *)
(* a function is just a sequence of length N. *)
(* We will also consider partial solutions, also represented as sequences *)
(* of length \leq N. *)
(***************************************************************************)

CONSTANT N \** number of queens and size of the board
ASSUME N \in Nat \ {0}
```

```

(* The following predicate determines if queens i and j attack each other
   in a placement of queens (represented by a sequence as above). *)
Attacks(queens,i,j) ==
  \/ queens[i] = queens[j] \** same column
  \/ queens[i] - queens[j] = i - j \** first diagonal
  \/ queens[j] - queens[i] = i - j \** second diagonal

(* A placement represents a (partial) solution if no two different queens
   attack each other in it. *)
IsSolution(queens) ==
  \A i \in 1 .. Len(queens)-1 : \A j \in i+1 .. Len(queens) :
       ~ Attacks(queens,i,j)

(* Compute the set of solutions of the N-queens problem. *)
Solutions == { queens \in [1..N -> 1..N] : IsSolution(queens) }

(***************************************************************************)
(* We now describe an algorithm that iteratively computes the set of *)
(* solutions of the N-queens problem by successively placing queens. *)
(* The current state of the algorithm is given by two variables: *)
(* - todo contains a set of partial solutions, *)
(* - sols contains the set of full solutions found so far. *)
(* At every step, the algorithm picks some partial solution and computes *)
(* all possible extensions by the next queen. If N queens have been placed *)
(* these extensions are in fact full solutions, otherwise they are added *)
(* to the set todo. *)
(***************************************************************************)

(* --algorithm QueensPluscal
     variables
       todo = { << >> };
       sols = {};

     begin
nxtQ: while todo # {}
       do
         with queens \in todo,
              nxtQ = Len(queens) + 1,
              cols = { c \in 1..N : ~ \E i \in 1 .. Len(queens) :
                                    Attacks( Append(queens, c), i, nxtQ ) },
              exts = { Append(queens,c) : c \in cols }
         do
           if (nxtQ = N)
           then todo := todo \ {queens}; sols := sols \union exts;
           else todo := (todo \ {queens}) \union exts;
           end if;
         end with;
       end while;
       print sols;
     end algorithm
*)

\** BEGIN TRANSLATION
VARIABLES todo, sols, pc

vars == << todo, sols, pc >>

Init == (* Global variables *)
        /\ todo = { << >> }
        /\ sols = {}
        /\ pc = "nxtQ"
```

```
nxtQ == /\ pc = "nxtQ"
        /\ IF todo # {}
              THEN /\ \E queens \in todo:
                        LET nxtQ == Len(queens) + 1 IN
                          LET cols == { c \in 1..N : ~ \E i \in 1 .. Len(queens) :
                                                     Attacks( Append(queens, c), i, nxtQ ) } IN
                            LET exts == { Append(queens,c) : c \in cols } IN
                              IF (nxtQ = N)
                                 THEN /\ todo' = todo \ {queens}
                                      /\ sols' = (sols \union exts)
                                 ELSE /\ todo' = ((todo \ {queens}) \union exts)
                                      /\ sols' = sols
                   /\ pc' = "nxtQ"
              ELSE /\ PrintT(sols)
                   /\ pc' = "Done"
                   /\ UNCHANGED << todo, sols >>

Next == nxtQ
           \/ (* Disjunct to prevent deadlock on termination *)
              (pc = "Done" /\ UNCHANGED vars)

Spec == Init /\ [][Next]_vars

Termination == <>(pc = "Done")

\** END TRANSLATION

TypeInvariant ==
  /\ todo \in SUBSET Seq(1 .. N) /\ \A s \in todo : Len(s) < N
  /\ sols \in SUBSET Seq(1 .. N) /\ \A s \in sols : Len(s) = N

(* The set of sols contains only solutions, and contains all solutions
   when todo is empty. *)
Invariant ==
  /\ sols \subseteq Solutions
  /\ todo = {} => Solutions \subseteq sols

(* Assert that no solutions are ever computed so that TLC displays one *)
NoSolutions == sols = {}

(* Add a fairness condition to ensure progress as long as todo is nonempty *)
Liveness == WF_vars(nxtQ)
LiveSpec == Spec /\ Liveness

=============================================================================
\* Modification History
\* Last modified Sat Jun 02 07:28:16 EDT 2018 by osboxes
\* Last modified Sat Dec 18 18:57:03 CET 2010 by merz
\* Created Sat Dec 11 08:50:24 CET 2010 by merz
```

PlusCal

```
package main

import (
    "fmt"
    "sort"
)

var N int
```

```

func init() {
    N = 8
}

func Attacks(queens []int, i int, j int) bool {
    return queens[i-1] == queens[j-1] || queens[i-1]-queens[j-1] == i-j || queens[j-1]-queens[i-1] == i-j
}

func main() {
    todo := [][]int{[]int{}}
    sols := [][]int{}
    for {
        if !(len(todo) != 0) {
            break
        }
        queens := todo[0]
        nxtQ := len(queens) + 1
        tmpSet := make([]int, 0)
        tmpRange := make([]int, N-1+1)
        for i := 1; i <= N; i++ {
            tmpRange[i-1] = i
        }
        for _, c := range tmpRange {
            exists := false
            tmpRange0 := make([]int, len(queens)-1+1)
            for i := 1; i <= len(queens); i++ {
                tmpRange0[i-1] = i
            }
            for _, i := range tmpRange0 {
                tmpSlice := make([]int, len(queens), len(queens)+1)
                copy(tmpSlice, queens)
                tmpSlice = append(tmpSlice, c)
                if Attacks(tmpSlice, i, nxtQ) {
                    exists = true
                    break
                }
            }
            if !exists {
                tmpSet = append(tmpSet, c)
            }
        }
        cols := tmpSet
        tmpSet0 := make([][]int, 0)
        for _, c := range cols {
            tmpSlice := make([]int, len(queens), len(queens)+1)
            copy(tmpSlice, queens)
            tmpSlice = append(tmpSlice, c)
            tmpSet0 = append(tmpSet0, tmpSlice)
        }
        sort.Slice(tmpSet0, func(i int, j int) bool {
            less := len(tmpSet0[i]) < len(tmpSet0[j])
            if len(tmpSet0[i]) == len(tmpSet0[j]) {
                for i0 := 0; i0 < len(tmpSet0[i]); i0++ {
                    less = tmpSet0[i][i0] < tmpSet0[j][i0]
                    if tmpSet0[i][i0] != tmpSet0[j][i0] {
                        break
                    }
                }
            }
            return less
        })
```

```
        if len(tmpSet0) > 1 {
            previousValue := tmpSet0[0]
            currentIndex := 1
            for _, v := range tmpSet0[1:] {
                eq := len(previousValue) == len(v)
                if eq {
                    for i0 := 0; i0 < len(previousValue); i0++ {
                        eq = previousValue[i0] == v[i0]
                        if !eq {
                            break
                        }
                    }
                }
                if !eq {
                    tmpSet0[currentIndex] = v
                    currentIndex++
                }
                previousValue = v
            }
            tmpSet0 = tmpSet0[:currentIndex]
        }
        exts := tmpSet0
        if nxtQ == N {
            tmpSet1 := make([][]int, 0, len(todo))
            for _, v := range todo {
                eq := len(v) == len(queens)
                if eq {
                    for i0 := 0; i0 < len(v); i0++ {
                        eq = v[i0] == queens[i0]
                        if !eq {
                            break
                        }
                    }
                }
                if !eq {
                    tmpSet1 = append(tmpSet1, v)
                }
            }
            todo = tmpSet1
            tmpSet2 := make([][]int, len(sols), len(sols)+len(exts))
            copy(tmpSet2, sols)
            tmpSet2 = append(tmpSet2, exts...)
            sort.Slice(tmpSet2, func(i0 int, j0 int) bool {
                less0 := len(tmpSet2[i0]) < len(tmpSet2[j0])
                if len(tmpSet2[i0]) == len(tmpSet2[j0]) {
                    for i1 := 0; i1 < len(tmpSet2[i0]); i1++ {
                        less0 = tmpSet2[i0][i1] < tmpSet2[j0][i1]
                        if tmpSet2[i0][i1] != tmpSet2[j0][i1] {
                            break
                        }
                    }
                }
                return less0
            })
            if len(tmpSet2) > 1 {
                previousValue := tmpSet2[0]
                currentIndex := 1
                for _, v := range tmpSet2[1:] {
                    eq := len(previousValue) == len(v)
                    if eq {
                        for i1 := 0; i1 < len(previousValue); i1++ {
                            eq = previousValue[i1] == v[i1]
```

```
                            if !eq {
                                break
                            }
                        }
                    }
                    if !eq {
                        tmpSet2[currentIndex] = v
                        currentIndex++
                    }
                    previousValue = v
                }
                tmpSet2 = tmpSet2[:currentIndex]
            }
            sols = tmpSet2
        } else {
            tmpSet1 := make([][]int, 0, len(todo))
            for _, v := range todo {
                eq := len(v) == len(queens)
                if eq {
                    for i0 := 0; i0 < len(v); i0++ {
                        eq = v[i0] == queens[i0]
                        if !eq {
                            break
                        }
                    }
                }
                if !eq {
                    tmpSet1 = append(tmpSet1, v)
                }
            }
            tmpSet2 := make([][]int, len(tmpSet1), len(tmpSet1)+len(exts))
            copy(tmpSet2, tmpSet1)
            tmpSet2 = append(tmpSet2, exts...)
            sort.Slice(tmpSet2, func(i0 int, j0 int) bool {
                less0 := len(tmpSet2[i0]) < len(tmpSet2[j0])
                if len(tmpSet2[i0]) == len(tmpSet2[j0]) {
                    for i1 := 0; i1 < len(tmpSet2[i0]); i1++ {
                        less0 = tmpSet2[i0][i1] < tmpSet2[j0][i1]
                        if tmpSet2[i0][i1] != tmpSet2[j0][i1] {
                            break
                        }
                    }
                }
                return less0
            })
            if len(tmpSet2) > 1 {
                previousValue := tmpSet2[0]
                currentIndex := 1
                for _, v := range tmpSet2[1:] {
                    eq := len(previousValue) == len(v)
                    if eq {
                        for i1 := 0; i1 < len(previousValue); i1++ {
                            eq = previousValue[i1] == v[i1]
                            if !eq {
                                break
                            }
                        }
                    }
                    if !eq {
                        tmpSet2[currentIndex] = v
                        currentIndex++
                    }
```

```go
                previousValue = v
            }
            tmpSet2 = tmpSet2[:currentIndex]
        }
        todo = tmpSet2
    }
}
fmt.Printf("%v\n", sols)
}
```

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.

```
--------------------------- MODULE DijkstraMutex ---------------------------
(***************************************************************************)
(* This is a PlusCal version of the first published mutual exclusion *)
(* algorithm, which appeared in *)
(* *)
(* E. W. Dijkstra: "Solution of a Problem in Concurrent *)
(* Programming Control". Communications of the ACM 8, 9 *)
(* (September 1965) page 569. *)
(* *)
(* Here is the description of the algorithm as it appeared in that paper. *)
(* The global variables are declared by *)
(* *)
(* Boolean array b, c[1:N]; integer k *)
(* *)
(* The initial values of b[i] and c[i] are true, for each i in 1..N. The *)
(* initial value of k can be any integer in 1..N. The pseudo-code for the *)
(* i-th process, for each i in 1..N, is: *)
(* *)
(* integer j; *)
(* Li0: b[i] := false; *)
(* Li1: if k # i then *)
(* Li2: begin c[i] := true; *)
(* Li3: if b[k] then k := i; *)
(* go to Li1 *)
(* end *)
(* else *)
(* Li4: begin c[i] := false; *)
(* for j := 1 step 1 until N do *)
(* if j # i and not c[j] then go to Li1 *)
(* end; *)
(* critical section; *)
(* c[i] := true; b[i] := true; *)
(* remainder of the cycle in which stopping is allowed; *)
(* go to Li0 *)
(* *)
(* It appears to me that the "else" preceding label Li4 begins the else *)
(* clause of the if statement beginning at Li1, and that the code from Li4 *)
(* through the end three lines later should be the body of that else *)
(* clause. However, the indentation indicates otherwise. Moreover, that *)
(* interpretation produces an incorrect algorithm. It seems that this *)
(* "else" actually marks an empty else clause for the if statement at Li1. *)
(* (Perhaps there should have been a semicolon after the "else".) *)
(***************************************************************************)

```

```
EXTENDS Integers

(******************************************************************************)
(* There is no reason why the processes need to be numbered from 1 to N. *)
(* So, we assume an arbitrary set Proc of process names. *)
(******************************************************************************)
CONSTANT Proc

(*********
Here is the PlusCal version of this algorithm.
The algorithm was modified from the original by adding a the variable temp2,
  to avoid a type consistency conflict when temp changes type at Li4a.

 --algorithm Mutex
 { variables b = [i \in Proc |-> TRUE], c = [i \in Proc |-> TRUE], k \in Proc;
   process (P \in Proc)
     variable temp, temp2 ;
     { Li0: while (TRUE)
            { b[self] := FALSE;
              Li1: if (k # self) { Li2: c[self] := TRUE;
                                Li3a: temp := k;
                                Li3b: if (b[temp]) { Li3c: k := self } ;
                                Li3d: goto Li1
                               };
             Li4a: c[self] := FALSE;
                   temp2 := Proc \ {self};
             Li4b: while (temp2 # {})
                    { with (j \in temp2)
                       { temp2 := temp2 \ {j};
                         if (~c[j]) { goto Li1 }
                       }
                    };
               cs: skip; \* the critical section
              Li5: c[self] := TRUE;
              Li6: b[self] := TRUE;
              ncs: skip \* non-critical section ("remainder of cycle")
            }
     }
 }
Notes on the PlusCal version:

1. Label Li3d is required by the translation. It could be eliminated by
   adding a then clause to the if statement of Li3b and putting the goto
   in both branches of the if statement.

2. The for loop in section Li4 of the original has been changed to
   a while loop that examines the other processes in an arbitrary
   (nondeterministically chosen) order. Because temp is set equal
   to the set of all processes other than self, there is no need for
   a test corresponding to the "if j # i" in the original. Note that
   the process-local variable j has been replaced by the identifier
   j that is local to the with statement.
*********)
\* BEGIN TRANSLATION
CONSTANT defaultInitValue
VARIABLES b, c, k, pc, temp

vars == << b, c, k, pc, temp >>

ProcSet == (Proc)

Init == (* Global variables *)
```

```
        /\ b = [i \in Proc |-> TRUE]
        /\ c = [i \in Proc |-> TRUE]
        /\ k \in Proc
        (* Process P *)
        /\ temp = [self \in Proc |-> defaultInitValue]
        /\ pc = [self \in ProcSet |-> CASE self \in Proc -> "Li0"]

Li0(self) == /\ pc[self] = "Li0"
             /\ b' = [b EXCEPT ![self] = FALSE]
             /\ pc' = [pc EXCEPT ![self] = "Li1"]
             /\ UNCHANGED << c, k, temp >>

Li1(self) == /\ pc[self] = "Li1"
             /\ IF k # self
                   THEN /\ pc' = [pc EXCEPT ![self] = "Li2"]
                   ELSE /\ pc' = [pc EXCEPT ![self] = "Li4a"]
             /\ UNCHANGED << b, c, k, temp >>

Li2(self) == /\ pc[self] = "Li2"
             /\ c' = [c EXCEPT ![self] = TRUE]
             /\ pc' = [pc EXCEPT ![self] = "Li3a"]
             /\ UNCHANGED << b, k, temp >>

Li3a(self) == /\ pc[self] = "Li3a"
              /\ temp' = [temp EXCEPT ![self] = k]
              /\ pc' = [pc EXCEPT ![self] = "Li3b"]
              /\ UNCHANGED << b, c, k >>

Li3b(self) == /\ pc[self] = "Li3b"
              /\ IF b[temp[self]]
                    THEN /\ pc' = [pc EXCEPT ![self] = "Li3c"]
                    ELSE /\ pc' = [pc EXCEPT ![self] = "Li3d"]
              /\ UNCHANGED << b, c, k, temp >>

Li3c(self) == /\ pc[self] = "Li3c"
              /\ k' = self
              /\ pc' = [pc EXCEPT ![self] = "Li3d"]
              /\ UNCHANGED << b, c, temp >>

Li3d(self) == /\ pc[self] = "Li3d"
              /\ pc' = [pc EXCEPT ![self] = "Li1"]
              /\ UNCHANGED << b, c, k, temp >>

Li4a(self) == /\ pc[self] = "Li4a"
              /\ c' = [c EXCEPT ![self] = FALSE]
              /\ temp' = [temp EXCEPT ![self] = Proc \ {self}]
              /\ pc' = [pc EXCEPT ![self] = "Li4b"]
              /\ UNCHANGED << b, k >>

Li4b(self) == /\ pc[self] = "Li4b"
              /\ IF temp[self] # {}
                    THEN /\ \E j \in temp[self]:
                              /\ temp' = [temp EXCEPT
                                            ![self] = temp[self] \ {j}]
                              /\ IF ~c[j]
                                    THEN /\ pc' = [pc EXCEPT
                                                    ![self] = "Li1"]
                                    ELSE /\ pc' = [pc EXCEPT
                                                    ![self] = "Li4b"]
                    ELSE /\ pc' = [pc EXCEPT ![self] = "cs"]
                         /\ UNCHANGED temp
              /\ UNCHANGED << b, c, k >>
```

```

cs(self) == /\ pc[self] = "cs"
           /\ TRUE
           /\ pc' = [pc EXCEPT ![self] = "Li5"]
           /\ UNCHANGED << b, c, k, temp >>

Li5(self) == /\ pc[self] = "Li5"
            /\ c' = [c EXCEPT ![self] = TRUE]
            /\ pc' = [pc EXCEPT ![self] = "Li6"]
            /\ UNCHANGED << b, k, temp >>

Li6(self) == /\ pc[self] = "Li6"
            /\ b' = [b EXCEPT ![self] = TRUE]
            /\ pc' = [pc EXCEPT ![self] = "ncs"]
            /\ UNCHANGED << c, k, temp >>

ncs(self) == /\ pc[self] = "ncs"
            /\ TRUE
            /\ pc' = [pc EXCEPT ![self] = "Li0"]
            /\ UNCHANGED << b, c, k, temp >>

P(self) == Li0(self) \/ Li1(self) \/ Li2(self) \/ Li3a(self) \/ Li3b(self)
             \/ Li3c(self) \/ Li3d(self) \/ Li4a(self) \/ Li4b(self)
             \/ cs(self) \/ Li5(self) \/ Li6(self) \/ ncs(self)

Next == (\E self \in Proc: P(self))
          \/ (* Disjunct to prevent deadlock on termination *)
             ((\A self \in ProcSet: pc[self] = "Done") /\ UNCHANGED vars)

Spec == Init /\ [][Next]_vars /\ \A self \in Proc: WF_vars(P(self))

Termination == <>(\A self \in ProcSet: pc[self] = "Done")

\* END TRANSLATION

(***************************************************************************)
(* The following formula asserts that no two processes are in their *)
(* critcal sections at the same time. It is the invariant that a mutual *)
(* exclusion algorithm should satisfy. You can have TLC check that the *)
(* algorithm is a mutual exclusion algorithm by checking that this formula *)
(* is an invariant. *)
(***************************************************************************)
MutualExclusion == \A i, j \in Proc :
                    (i # j) => ~ /\ pc[i] = "cs"
                                 /\ pc[j] = "cs"
(***************************************************************************)
(* An equivalent way to perform the same test would be to change the *)
(* statement labeled cs (the critical section) to *)
(* *)
(* cs: assert \A j \in Proc \ {self} : pc[j] # "cs" *)
(* *)
(* You can give this a try. However, the assert statement requires that *)
(* the EXTENDS statement also import the standard module TLC, so it should *)
(* read *)
(* *)
(* EXTENDS Integers, TLC *)
(***************************************************************************)

(***************************************************************************)
(* LIVENESS *)
(* *)
(* If you are a sophisticated PlusCal user and know a little temporal *)
```

```
(* logic, you can continue reading about the liveness properties of the *)
(* algorithm. *)
(* *)
(* Dijkstra's algorithm is "deadlock free", which for a mutual exclusion *)
(* algorithm means that if some process is trying to enter its critical *)
(* section, then some process (not necessarily the same one) will *)
(* eventually enter its critical section. Since a process begins trying *)
(* to enter its critical section when it is at the control point labeled *)
(* Li0, and it is in its critical section when it is at control point cs, *)
(* the following formula asserts deadlock freedom. *)
(***************************************************************************)
DeadlockFree == \A i \in Proc :
                    (pc[i] = "Li0") ~> (\E j \in Proc : pc[j] = "cs")
(***************************************************************************)
(* Dijkstra's algorithm is deadlock free only under the assumption of *)
(* fairness of process execution. The simplest such fairness assumption *)
(* is weak fairness on each process's next-state action. This means that *)
(* no process can halt if it is always possible for that process to take a *)
(* step. The following statement tells the PlusCal translator to define *)
(* the specification to assert weak fairness of each process's next-state *)
(* action. *)
(* *)
(* PlusCal options (wf) *)
(* *)
(* This statement can occur anywhere in the file--either in a comment or *)
(* before or after the module. Because of this statement, the translator *)
(* has added the necessary fairness conjunct to the definition of Spec. *)
(* So, you can have the TLC model checker check that the algorithm *)
(* satisfies property DeadlockFree. *)
(***************************************************************************)

(***************************************************************************)
(* Dijkstra's algorithm is not "starvation free", because it allows some *)
(* waiting processes to "starve", never entering their critical section *)
(* while other processes keep entering and leaving their critical *)
(* sections. Starvation freedom is asserted by the following formula. *)
(* You can use TLC to show that the algorithm is not starvation free by *)
(* producing a counterexample trace. *)
(***************************************************************************)
StarvationFree == \A i \in Proc :
                    (pc[i] = "Li0") ~> (pc[i] = "cs")

(***************************************************************************)
(* In this algorithm, no process can ever be blocked waiting at an 'await' *)
(* statement or a 'with (v \in S)' statement with S empty. Therefore, *)
(* weak fairness of each process means that each process keeps continually *)
(* trying to enter its critical section, and it exits the critical *)
(* section. An important requirement of a mutual exclusion solution, one *)
(* that rules out many simple solutions, is that a process is allowed to *)
(* remain forever in its non-critical section. (There is also no need to *)
(* require that a process that enters its critical section ever leaves it, *)
(* though without that requirement the definition of starvation freedom *)
(* must be changed.) *)
(* *)
(* We can allow a process to remain forever in its critical section by *)
(* replacing the 'skip' statement that represents the non-critical section *)
(* with the following statement, which allows the process to loop forever. *)
(* *)
(* ncs: either skip or goto ncs *)
(* *)
(* An equivalent non-critical section is *)
(* *)
```

```
(* nsc: either skip or await FALSE *)
(* *)
(* A more elegant method is to change the fairness requirement to assert *)
(* weak fairness of a process's next-state action only when the process is *)
(* not in its non-critical section. This is accomplished by taking the *)
(* following formula LSpec as the algorithm's specification. *)
(***************************************************************************)
LSpec == Init /\ [][Next]_vars
          /\ \A self \in Proc: WF_vars((pc[self] # "ncs") /\ P(self))

(***************************************************************************)
(* If we allow a process to remain forever in its non-critical section, *)
(* then our definition of deadlock freedom is too weak. Suppose process p *)
(* were in its critical section and process q, trying to enter its *)
(* critical section, reached Li1. Formula DeadlockFree would allow a *)
(* behavior in which process q exited its critical section and remained *)
(* forever in its non-critical section, but process p looped forever *)
(* trying to enter its critical section and never succeeding. To rule out *)
(* this possibility, we must replace the formula *)
(* *)
(* pc[i] = "Li0" *)
(* *)
(* in DeadLock free with one asserting that control in process i is *)
(* anywhere in control points Li0 through Li4b. It's easier to express *)
(* this by saying where control in process i is NOT, which we do in the *)
(* following property. *)
(***************************************************************************)
DeadlockFreedom ==
    \A i \in Proc :
      (pc[i] \notin {"Li5", "Li6", "ncs"}) ~> (\E j \in Proc : pc[j] = "cs")
(***************************************************************************)
(* Do you see why it's not necessary to include "cs" in the set of values *)
(* that pc[i] does not equal? *)
(***************************************************************************)



(***************************************************************************)
(* Using a single worker thread on a 2.5GHz dual-processor computer, TLC *)
(* can check MutualExclusion and liveness of a 3-process model in about 2 *)
(* or 3 minutes (depending on which spec is used and which liveness *)
(* property is checked). That model has 90882 reachable states and a *)
(* state graph of diameter 54. TLC can check a 4-process model in about *)
(* 53 minutes. That model has 33288512 reachable states and a state graph *)
(* of diameter 89. *)
(***************************************************************************)
=============================================================================
\* Modification History
\* Last modified Sun Dec 31 22:04:29 EST 2017 by osboxes
\* Last modified Sat Jan 01 12:14:14 PST 2011 by lamport
\* Created Fri Dec 31 14:14:14 PST 2010 by lamport
```

PlusCal

```
package main

import (
    "sort"
    "sync"
)

```

```go
var Proc []int

var b []struct {
	key int
	value bool
}

var c []struct {
	key int
	value bool
}

var k int

var pGoStart chan bool

var pGoWait sync.WaitGroup

var pGoLock []sync.RWMutex

func init() {
	tmpRange := make([]int, 3-1+1)
	for i := 1; i <= 3; i++ {
		tmpRange[i-1] = i
	}
	Proc = tmpRange
	function := make([]struct {
		key int
		value bool
	}, 0, len(Proc))
	for _, i := range Proc {
		function = append(function, struct {
			key int
			value bool
		}{key: i, value: true})
	}
	b = function
	function0 := make([]struct {
		key int
		value bool
	}, 0, len(Proc))
	for _, i := range Proc {
		function0 = append(function0, struct {
			key int
			value bool
		}{key: i, value: true})
	}
	c = function0
	k = Proc[0]
	pGoStart = make(chan bool)
	pGoLock = make([]sync.RWMutex, 3)
}

func P(self int) {
	defer pGoWait.Done()
	<-pGoStart
	temp := 0
	temp2 := []int{}
	pGoLock[2].Lock()
	for {
		if !true {
			pGoLock[2].Unlock()
```

```
            break
        }
        key := self
        index := sort.Search(len(b), func(i int) bool {
            return !(b[i].key < key)
        })
        b[index].value = false
        pGoLock[2].Unlock()
    Li1:
        pGoLock[1].Lock()
        if k != self {
            pGoLock[1].Unlock()
            pGoLock[0].Lock()
            key0 := self
            index0 := sort.Search(len(c), func(i0 int) bool {
                return !(c[i0].key < key0)
            })
            c[index0].value = true
            pGoLock[0].Unlock()
            pGoLock[1].Lock()
            temp = k
            pGoLock[1].Unlock()
            pGoLock[2].Lock()
            key1 := temp
            index1 := sort.Search(len(b), func(i1 int) bool {
                return !(b[i1].key < key1)
            })
            if b[index1].value {
                pGoLock[2].Unlock()
                pGoLock[1].Lock()
                k = self
                pGoLock[1].Unlock()
            } else {
                pGoLock[2].Unlock()
            }
            goto Li1
        } else {
            pGoLock[1].Unlock()
        }
        pGoLock[0].Lock()
        key0 := self
        index0 := sort.Search(len(c), func(i0 int) bool {
            return !(c[i0].key < key0)
        })
        c[index0].value = false
        tmpSet := make([]int, 0, len(Proc))
        for _, v := range Proc {
            if v != self {
                tmpSet = append(tmpSet, v)
            }
        }
        temp2 = tmpSet
        pGoLock[0].Unlock()
        pGoLock[0].Lock()
        for {
            if !(len(temp2) != 0) {
                break
            }
            j := temp2[0]
            tmpSet0 := make([]int, 0, len(temp2))
            for _, v := range temp2 {
                if v != j {
```

```
                tmpSet0 = append(tmpSet0, v)
            }
        }
        temp2 = tmpSet0
        key1 := j
        index1 := sort.Search(len(c), func(i1 int) bool {
            return !(c[i1].key < key1)
        })
        if !c[index1].value {
            pGoLock[0].Unlock()
            goto Li1
        }
    }
    pGoLock[0].Unlock()
    pGoLock[0].Lock()
    key1 := self
    index1 := sort.Search(len(c), func(i1 int) bool {
        return !(c[i1].key < key1)
    })
    c[index1].value = true
    pGoLock[0].Unlock()
    pGoLock[2].Lock()
    key2 := self
    index2 := sort.Search(len(b), func(i2 int) bool {
        return !(b[i2].key < key2)
    })
    b[index2].value = true
    pGoLock[2].Unlock()
    pGoLock[2].Lock()
  }
  pGoLock[2].Unlock()
}

func main() {
  for _, v := range Proc {
    pGoWait.Add(1)
    go P(v)
  }
  close(pGoStart)
  pGoWait.Wait()
}
```

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.

```
---------------------------- MODULE load_balancer ----------------------------
(***************************************************************************)
(* Specifies a simple load balancer. *)
(***************************************************************************)

\* Extends some built-in TLA+ modules
EXTENDS Naturals, Sequences, TLC

\* The `TCPChannel` mapping macro used in this specification is parameterized
\* by a `BUFFER_SIZE` constant. This value controls the number of messages being
```

```
\* held in a buffer by each process. Processes trying to send a message to another
\* process with a full buffer wil "block" (not be scheduled by TLC).
CONSTANT BUFFER_SIZE

\* Define a constant identifier for the load balancer.
LoadBalancerId == 0

\* The number of servers and clients in the model checking setup.
CONSTANTS NUM_SERVERS, NUM_CLIENTS

\* TLC should assume that both numbers are greater than zero (i.e., we always
\* have at least one server and one client). Note, however, that increasing
\* these numbers makes the number of states to be checked by TLC to grow
\* exponentially.
ASSUME NUM_SERVERS > 0 /\ NUM_CLIENTS > 0

\* GET_PAGE is a label attached to messages sent from the clients to
\* the load balancer.
\*
\* WEB_PAGE abstractly represents a web page that the server may return
\* to clients. The content of the webpage is, obviously, orthogonal to the
\* correctness of our load balancer.
CONSTANTS GET_PAGE, WEB_PAGE

\* total nodes in the system:
\* number of clients + number of servers + the load balancer
NUM_NODES == NUM_CLIENTS + NUM_SERVERS + 1

(***************************************************************************
--mpcal LoadBalancer {

  \* The TCPChannel mapping macro models a communication channel
  \* between two processes using TCP-like semantics. In particular:
  \*
  \* - reading from the channel "blocks" unless there is a message
  \* ready to be read.
  \* - message delivery is reliable and ordered (i.e., FIFO).
  mapping macro TCPChannel {
      read {
          await Len($variable) > 0;
          with (msg = Head($variable)) {
              $variable := Tail($variable);
              yield msg;
          };
      }

      write {
          await Len($variable) < BUFFER_SIZE;
          yield Append($variable, $value);
      }
  }

  \* Mapping macros keep implementation-specific behavior that we don't
  \* want to model check outside of our archetype definitions.
  \* In the case of this load balancer, how a server retrieves a web page
  \* is orthogonal to the correctness of the properties we are interested
  \* to check with this specification.
  \*
  \* This mapping macro abstracts the process of reading a web page by
  \* always returning the 'WEB_PAGE' constant when the variable is read.
  \*
  \* Since "writing" to this mapping is meaningless, the attempting to write
```

```
\* to a variable mapped with WebPage will result in a model checking
\* error (see `assert(FALSE)` in the write definition).
mapping macro WebPages {
   read {
       yield WEB_PAGE;
   }

   write {
       assert(FALSE);
       yield $value;
   }
}

\* ALoadBalancer is the archetype that defines the behavior of
\* the load balancer process. The `mailboxes` parameter represents
\* connections to all nodes in the system.
archetype ALoadBalancer(ref mailboxes)

\* Local variables of this archetype:
variables
          \* Holds messages received by the load balancer (sent
          \* by clients)
          msg,

          \* identifier attached to every message sent to servers by
          \* the load balancer.
          next = 0;
{
    main:
      while (TRUE) {

         \* waits for a message to be received. Upon receipt, the `assert`
         \* call ensures that the message is of type `GET_PAGE`, the only
         \* type of message supported in this simple specification.
         \*
         \* Every message received by the load balancer is expected to
         \* be a tuple in the format:
         \*
         \* << {message_type}, {client_id} >>
         \*
         \* Note that tuples are 1-indexed.
         rcvMsg:
           msg := mailboxes[LoadBalancerId];
           assert(msg[1] = GET_PAGE);

         \* the load balancer needs to forward the client request to the
         \* server, who will process the request and send a web page back to
         \* the client.
         \*
         \* The message sent to the server is a tuple in the format:
         \*
         \* << {message_id}, {client_id} >>
         \*
         \* We send the client ID here so that the server can directly
         \* reply to a client, bypassing the load balancer. This is usually
         \* not what happens in practice, but the model is simple
         \* enough for our (illustrative) purposes.
         sendServer:
           next := (next % NUM_SERVERS) + 1;
           mailboxes[next] := << next, msg[2] >>;
      }
}
```

```

\* AServer is the archetype that defines the behavior of the servers
\* in our system. The two parameters it receives are:
\*
\* - mailboxes: contains connections to every node in the system
\* - page_stream: a source of web pages for the server. In practice,
\* this is implementation specific and irrelevant for
\* the properties we want to check in this specification
archetype AServer(ref mailboxes, page_stream)

\* Local variables
variable
         \* temporary buffer to hold incoming messages
         msg;
{
    serverLoop:
      while (TRUE) {

          \* waits for an incoming message. Note that the only process
          \* that sends messages to the server is the load balancer process
          \* (defined according to the ALoadBalancer archetype) and the
          \* message has the format << {message_id}, {client_id} >>
          rcvReq:
            msg := mailboxes[self];

        sendPage:
          \* sends a web page (read from ‘page_stream‘) back to the requester
          \* i.e., the client.
          mailboxes[msg[2]] := page_stream;
      }
}

\* GLOBAL VARIABLES *\

variables
          \* our network is modeled as a function from node identifier
          \* to a sequence of incoming messages
          network = [id \in 0..NUM_NODES |-> <<>>],

          \* the stream of web pages to be served by the server. Since we
          \* intend to map this variable using the WebPages mapping macro,
          \* the initial value assigned to it here is irrelevant.
          stream = 0;

\* PROCESS INSTANTIATION *\

\* The system has a single load balancer entity, instantiated from the ALoadBalancer
\* archetype. The model of our network is going to be the one defined by the TCPChannel
\* mapping macro in all instantiations.
fair process (LoadBalancer = LoadBalancerId) == instance ALoadBalancer(ref network)
    mapping network[_] via TCPChannel;

\* Instantiate ‘NUM_SERVERS‘ server processes according to the AServer archetype.
\* We map the page stream according to the WebPages mapping macro since this is
\* an implementation detail that needs to be specified during implementation at
\* a later stage.
fair process (Servers \in 1..NUM_SERVERS) == instance AServer(ref network, stream)
    mapping network[_] via TCPChannel
    mapping stream via WebPages;

\* We have a load balancer that waits for messages and servers that are ready
\* to serve web pages when the load balancer requests. However, there is one
```

```
  \* piece missing from this: a _client_ that actually drives the other two
  \* components.
  \*
  \* To illustrate how regular PlusCal processes can be used with Modular PlusCal,
  \* we model the client as a vanilla PlusCal process.



  \* Client processes are given integer identifiers starting from NUM_SERVERS+1.
  \* Keep in mind that this "range" notation in PlusCal defines a set, and is
  \* _inclusive_ (i.e., NUM_SERVERS+NUM_CLIENTS+1 is part of the set).
  \*
  \* Also note that since the client needs to send network messages to processes
  \* defined by the previous archetypes, we need to have the exact same network
  \* model here. However, since mapping macros are not available for regular
  \* PlusCal processes, we need to copy the specification of the TCPChannel
  \* mapping macro in this client definition.
  fair process (Client \in (NUM_SERVERS+1)..(NUM_SERVERS+NUM_CLIENTS+1))

  \* Local variables
  variable
          \* Temporary buffer to hold incoming messages
          msg;
  {
      clientLoop:
        while (TRUE) {

          \* First, the client makes a request to the load balancer.
          \* The format of the message is a tuple: << {message_type}, {client_id} >>.
          \* If you check the ALoadBalancer definition, this is the message format
          \* expected there.
          \*
          \* Remember that `self` is an implicitly defined, immutable variable that
          \* contains the process identifier of the "running" process.
          clientRequest:
            msg := << GET_PAGE, self >>;
            await Len(network[LoadBalancerId]) < BUFFER_SIZE;
            network[LoadBalancerId] := Append(network[LoadBalancerId], msg);


          \* Clients then wait for the response to the previously sent request.
          \* Since there is only one type of web page in this simple specification
          \* (defined by the WEB_PAGE constant), we assert here that the message
          \* received indeed is equal our expected web page.
          clientReceive:
            await Len(network[self]) > 0;
              with (m = Head(network[self])) {
                  network[self] := Tail(network[self]);
                  assert(m = WEB_PAGE);
              };
        }
  }
}

\* BEGIN PLUSCAL TRANSLATION
--algorithm LoadBalancer {
    variables network = [id \in (0)..(NUM_NODES) |-> <<>>], stream = 0;
    fair process (LoadBalancer = LoadBalancerId)
    variables msg, next = 0, mailboxesRead, mailboxesWrite, mailboxesWrite0;
    {
        main:
            if (TRUE) {
```

```
            rcvMsg:
                await (Len(network[LoadBalancerId]))>(0);
                with (msg0 = Head(network[LoadBalancerId])) {
                    mailboxesWrite := [network EXCEPT ![LoadBalancerId] = Tail(network[LoadBalancerId])];
                    mailboxesRead := msg0;
                };
                msg := mailboxesRead;
                assert (msg[1])=(GET_PAGE);
                network := mailboxesWrite;

            sendServer:
                next := ((next)%(NUM_SERVERS))+(1);
                await (Len(network[next]))<(BUFFER_SIZE);
                mailboxesWrite := [network EXCEPT ![next] = Append(network[next], <<next, msg[2]>>)];
                network := mailboxesWrite;
                goto main;

        } else {
            mailboxesWrite0 := network;
            network := mailboxesWrite0;
        };

    }
    fair process (Servers \in (1)..(NUM_SERVERS))
    variables msg, mailboxesRead0, mailboxesWrite1, page_streamRead, mailboxesWrite2;
    {
        serverLoop:
        if (TRUE) {
            rcvReq:
                await (Len(network[self]))>(0);
                with (msg1 = Head(network[self])) {
                    mailboxesWrite1 := [network EXCEPT ![self] = Tail(network[self])];
                    mailboxesRead0 := msg1;
                };
                msg := mailboxesRead0;
                network := mailboxesWrite1;

            sendPage:
                page_streamRead := WEB_PAGE;
                await (Len(network[msg[2]]))<(BUFFER_SIZE);
                mailboxesWrite1 := [network EXCEPT ![msg[2]] = Append(network[msg[2]], page_streamRead)];
                network := mailboxesWrite1;
                goto serverLoop;

        } else {
            mailboxesWrite2 := network;
            network := mailboxesWrite2;
        };

    }
    fair process (Client \in ((NUM_SERVERS)+(1))..(((NUM_SERVERS)+(NUM_CLIENTS))+(1)))
    variables msg;
    {
        clientLoop:
        while (TRUE) {
            clientRequest:
                msg := <<GET_PAGE, self>>;
                await (Len(network[LoadBalancerId]))<(BUFFER_SIZE);
                network[LoadBalancerId] := Append(network[LoadBalancerId], msg);

            clientReceive:
                await (Len(network[self]))>(0);
```

```
                with (m = Head(network[self])) {
                    network[self] := Tail(network[self]);
                    assert (m)=(WEB_PAGE);
                };

        };

    }
}
\* END PLUSCAL TRANSLATION



***************************************************************************)
\* BEGIN TRANSLATION
\* Process variable msg of process LoadBalancer at line 255 col 15 changed to msg_
\* Process variable msg of process Servers at line 283 col 15 changed to msg_S
CONSTANT defaultInitValue
VARIABLES network, stream, pc, msg_, next, mailboxesRead, mailboxesWrite,
          mailboxesWrite0, msg_S, mailboxesRead0, mailboxesWrite1,
          page_streamRead, mailboxesWrite2, msg

vars == << network, stream, pc, msg_, next, mailboxesRead, mailboxesWrite,
           mailboxesWrite0, msg_S, mailboxesRead0, mailboxesWrite1,
           page_streamRead, mailboxesWrite2, msg >>

ProcSet == {LoadBalancerId} \cup ((1)..(NUM_SERVERS)) \cup (((NUM_SERVERS)+(1))..(((NUM_SERVERS)+(NUM_CLIENTS))+(1)))

Init == (* Global variables *)
        /\ network = [id \in (0)..(NUM_NODES) |-> <<>>]
        /\ stream = 0
        (* Process LoadBalancer *)
        /\ msg_ = defaultInitValue
        /\ next = 0
        /\ mailboxesRead = defaultInitValue
        /\ mailboxesWrite = defaultInitValue
        /\ mailboxesWrite0 = defaultInitValue
        (* Process Servers *)
        /\ msg_S = [self \in (1)..(NUM_SERVERS) |-> defaultInitValue]
        /\ mailboxesRead0 = [self \in (1)..(NUM_SERVERS) |-> defaultInitValue]
        /\ mailboxesWrite1 = [self \in (1)..(NUM_SERVERS) |-> defaultInitValue]
        /\ page_streamRead = [self \in (1)..(NUM_SERVERS) |-> defaultInitValue]
        /\ mailboxesWrite2 = [self \in (1)..(NUM_SERVERS) |-> defaultInitValue]
        (* Process Client *)
        /\ msg = [self \in ((NUM_SERVERS)+(1))..(((NUM_SERVERS)+(NUM_CLIENTS))+(1)) |-> defaultInitValue]
        /\ pc = [self \in ProcSet |-> CASE self = LoadBalancerId -> "main"
                                        [] self \in (1)..(NUM_SERVERS) -> "serverLoop"
                                        [] self \in ((NUM_SERVERS)+(1))..(((NUM_SERVERS)+(NUM_CLIENTS))+(1)) -> "clientLoo

main == /\ pc[LoadBalancerId] = "main"
        /\ IF TRUE
              THEN /\ pc' = [pc EXCEPT ![LoadBalancerId] = "rcvMsg"]
                   /\ UNCHANGED << network, mailboxesWrite0 >>
              ELSE /\ mailboxesWrite0' = network
                   /\ network' = mailboxesWrite0'
                   /\ pc' = [pc EXCEPT ![LoadBalancerId] = "Done"]
        /\ UNCHANGED << stream, msg_, next, mailboxesRead, mailboxesWrite,
                        msg_S, mailboxesRead0, mailboxesWrite1,
                        page_streamRead, mailboxesWrite2, msg >>

rcvMsg == /\ pc[LoadBalancerId] = "rcvMsg"
          /\ (Len(network[LoadBalancerId]))>(0)
```

```
        /\ LET msg0 == Head(network[LoadBalancerId]) IN
            /\ mailboxesWrite' = [network EXCEPT ![LoadBalancerId] = Tail(network[LoadBalancerId])]
            /\ mailboxesRead' = msg0
        /\ msg_' = mailboxesRead'
        /\ Assert((msg_'[1])=(GET_PAGE),
                "Failure of assertion at line 266, column 21.")
        /\ network' = mailboxesWrite'
        /\ pc' = [pc EXCEPT ![LoadBalancerId] = "sendServer"]
        /\ UNCHANGED << stream, next, mailboxesWrite0, msg_S, mailboxesRead0,
                        mailboxesWrite1, page_streamRead, mailboxesWrite2,
                        msg >>

sendServer == /\ pc[LoadBalancerId] = "sendServer"
           /\ next' = ((next)%(NUM_SERVERS))+(1)
           /\ (Len(network[next']))<(BUFFER_SIZE)
           /\ mailboxesWrite' = [network EXCEPT ![next'] = Append(network[next'], <<next', msg_[2]>>)]
           /\ network' = mailboxesWrite'
           /\ pc' = [pc EXCEPT ![LoadBalancerId] = "main"]
           /\ UNCHANGED << stream, msg_, mailboxesRead, mailboxesWrite0,
                           msg_S, mailboxesRead0, mailboxesWrite1,
                           page_streamRead, mailboxesWrite2, msg >>

LoadBalancer == main \/ rcvMsg \/ sendServer

serverLoop(self) == /\ pc[self] = "serverLoop"
                /\ IF TRUE
                       THEN /\ pc' = [pc EXCEPT ![self] = "rcvReq"]
                           /\ UNCHANGED << network, mailboxesWrite2 >>
                       ELSE /\ mailboxesWrite2' = [mailboxesWrite2 EXCEPT ![self] = network]
                           /\ network' = mailboxesWrite2'[self]
                           /\ pc' = [pc EXCEPT ![self] = "Done"]
                /\ UNCHANGED << stream, msg_, next, mailboxesRead,
                                mailboxesWrite, mailboxesWrite0, msg_S,
                                mailboxesRead0, mailboxesWrite1,
                                page_streamRead, msg >>

rcvReq(self) == /\ pc[self] = "rcvReq"
            /\ (Len(network[self]))>(0)
            /\ LET msg1 == Head(network[self]) IN
                /\ mailboxesWrite1' = [mailboxesWrite1 EXCEPT ![self] = [network EXCEPT ![self] = Tail(network[self
                /\ mailboxesRead0' = [mailboxesRead0 EXCEPT ![self] = msg1]
            /\ msg_S' = [msg_S EXCEPT ![self] = mailboxesRead0'[self]]
            /\ network' = mailboxesWrite1'[self]
            /\ pc' = [pc EXCEPT ![self] = "sendPage"]
            /\ UNCHANGED << stream, msg_, next, mailboxesRead,
                            mailboxesWrite, mailboxesWrite0,
                            page_streamRead, mailboxesWrite2, msg >>

sendPage(self) == /\ pc[self] = "sendPage"
              /\ page_streamRead' = [page_streamRead EXCEPT ![self] = WEB_PAGE]
              /\ (Len(network[msg_S[self][2]]))<(BUFFER_SIZE)
              /\ mailboxesWrite1' = [mailboxesWrite1 EXCEPT ![self] = [network EXCEPT ![msg_S[self][2]] = Append(net
              /\ network' = mailboxesWrite1'[self]
              /\ pc' = [pc EXCEPT ![self] = "serverLoop"]
              /\ UNCHANGED << stream, msg_, next, mailboxesRead,
                              mailboxesWrite, mailboxesWrite0, msg_S,
                              mailboxesRead0, mailboxesWrite2, msg >>

Servers(self) == serverLoop(self) \/ rcvReq(self) \/ sendPage(self)

clientLoop(self) == /\ pc[self] = "clientLoop"
                /\ pc' = [pc EXCEPT ![self] = "clientRequest"]
```

```
                  /\ UNCHANGED << network, stream, msg_, next, mailboxesRead,
                                  mailboxesWrite, mailboxesWrite0, msg_S,
                                  mailboxesRead0, mailboxesWrite1,
                                  page_streamRead, mailboxesWrite2, msg >>

clientRequest(self) == /\ pc[self] = "clientRequest"
                       /\ msg' = [msg EXCEPT ![self] = <<GET_PAGE, self>>]
                       /\ (Len(network[LoadBalancerId]))<(BUFFER_SIZE)
                       /\ network' = [network EXCEPT ![LoadBalancerId] = Append(network[LoadBalancerId], msg'[self])]
                       /\ pc' = [pc EXCEPT ![self] = "clientReceive"]
                       /\ UNCHANGED << stream, msg_, next, mailboxesRead,
                                       mailboxesWrite, mailboxesWrite0, msg_S,
                                       mailboxesRead0, mailboxesWrite1,
                                       page_streamRead, mailboxesWrite2 >>

clientReceive(self) == /\ pc[self] = "clientReceive"
                       /\ (Len(network[self]))>(0)
                       /\ LET m == Head(network[self]) IN
                            /\ network' = [network EXCEPT ![self] = Tail(network[self])]
                            /\ Assert((m)=(WEB_PAGE),
                                      "Failure of assertion at line 323, column 25.")
                       /\ pc' = [pc EXCEPT ![self] = "clientLoop"]
                       /\ UNCHANGED << stream, msg_, next, mailboxesRead,
                                       mailboxesWrite, mailboxesWrite0, msg_S,
                                       mailboxesRead0, mailboxesWrite1,
                                       page_streamRead, mailboxesWrite2, msg >>

Client(self) == clientLoop(self) \/ clientRequest(self)
                 \/ clientReceive(self)

Next == LoadBalancer
          \/ (\E self \in (1)..(NUM_SERVERS): Servers(self))
          \/ (\E self \in ((NUM_SERVERS)+(1))..(((NUM_SERVERS)+(NUM_CLIENTS))+(1)): Client(self))

Spec == /\ Init /\ [][Next]_vars
        /\ WF_vars(LoadBalancer)
        /\ \A self \in (1)..(NUM_SERVERS) : WF_vars(Servers(self))
        /\ \A self \in ((NUM_SERVERS)+(1))..(((NUM_SERVERS)+(NUM_CLIENTS))+(1)) : WF_vars(Client(self))

\* END TRANSLATION


(* INVARIANTS *)


\* This is an _invariant_ of our specification: in other words,
\* we expect the BuffersOk predicate to always be true in every step of execution
BufferOk(node) == Len(network[node]) >= 0 /\ Len(network[node]) <= BUFFER_SIZE
BuffersOk == \A node \in DOMAIN network : BufferOk(node)


(* PROPERTIES *)

\* This is a property we would like to check about our specification.
\* Properties are defined using _temporal logic_. In this specific example,
\* we want to make sure that every client that requests a web page (i.e., are
\* in the 'clientRequest' label) eventually receive a response (i.e., are
\* in the 'clientReceive' label). In order to specify this property, we have to
\* write the formula as if the client enters 'clientReceive' label, it will
\* eventually successfully receive a response and then go back to issuing
\* another request in the 'clientRequest' label.
ReceivesPage(client) == pc[client] = "clientReceive" ~> pc[client] = "clientRequest"
```

```
ClientsOk == \A client \in (NUM_SERVERS+1)..(NUM_SERVERS+NUM_CLIENTS+1) : ReceivesPage(client)

================================================================================
\* Modification History
\* Last modified Wed Feb 06 17:26:45 PST 2019 by minh
\* Last modified Sat Feb 02 19:59:51 PST 2019 by rmc
```

Modular PlusCal