PGo Manual
For commit 06215af5

August 29, 2018

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 is not a programming language so it cannot be run, which limits its utility.

PGo aims to correspond a verified PlusCal specification with an executable Go implementation. PGo compiles an annotated PlusCal specification into a Go program which preserves the semantics of the specification. The main goal of PGo is to create a compiled Go program which is easy to read and edit. The intended use case for PGo is for the developer to create and verify a PlusCal spec of the system, compile it with PGo, then edit the compiled program to fully implement details not included in the specification. Programs compiled by PGo are type safe (they do not contain type assertions which panic).

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 pgo` assuming the project is in the `pgo/` directory

Dependencies:

- The Plume options library.

- Java Hamcrest.

- The JSON reference implementation.

PGo was tested on JRE 8 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:

```
-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]
-c --configFilePath=<string> path to the configuration file, if any [default ]
--writeAST=<boolean> write the AST generated and skip the rest [default false]
```

2.3 Configuration

PGo requires a JSON configuration file with the following information.

```
1  {
2      "build": {
3          "output_dir": "",
4          "dest_file": ""
5      },
6      "networking": {
7          "enabled": true,
8          "state": {
9              "strategy": "",
10             "endpoints": [],
11             "peers": [],
12             "timeout": 3
13         }
14     },
15     "constants": {
16         "name": "value"
17     }
18 }
```

2.3.1 Build

`output_dir` must point to an existing directory.

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

2.3.2 Networking

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

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

`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 (e.g., constant N for model checking). These variables will need to be specified as key-value pairs. Each key is a JSON string containing the name of the constant being defined. Each value is a JSON string containing one valid TLA+ expression.

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

Example constant specification

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

Compiled Go

2.4 Type inference

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

2.5 Lock inference

PGo addds 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

Below are the PlusCal constructs. 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
case1:
stmt3
stmt4
case2:
stmt5
stmt6
// ...
endEither:```
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
Skip statement	`skip;`	Supported; compiled to nothing
Call statement	`call procedure1(arg1, arg2);`	Supported; compiled as expected
Macro call	`macro1(arg1, arg2);`	Supported; expanded during compilation
With statement	```with (x = exp1, y \in exp2) {
body;
}```	Supported; compiled as variable assignment with fresh names
Print statement	`print exp;`	Supported; compiled as expected
Assert statement	`assert condition;`	Compiled as ```if !condition {
panic("condition");
}```

Await statement	`await condition;`	Compiled as ```\nawaitLabel:\n if !condition {\n goto awaitLabel\n }\n```
Goto statement	`goto label;`	Supported; compiled as expected
Single process algorithm	```\n--algorithm Algo {\n variables x = exp1, y \in exp2;\n {\n body;\n }\n}\n```	Supported; compiled as a single-threaded single-process Go program
Multiprocess algorithm	```\n--algorithm Algo {\n variables x = exp1, y = exp2;\n process (P \in exp3)\n variables local = exp4;\n {\n body;\n }\n}\n```	Supported; compiled with various strategies configured by the user

PlusCal constructs

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

TLA+ feature	Example code	PGo support
Function call	```\nx[exp1]\n* or\nx[exp1, exp2, exp3]\n* or\nx[<<exp1, exp2, exp3>>]\n* or\nx[[field1 |-> e1, field2 |-> e2]]\n```	Supported; compiled code dependent on the type of `x` (the function)
Binary operator call	`x /\ y = z + 1`	Supported; compiled as expected
Function (as struct literal)	```\n[field1 \in exp1 |-> exp2,\n field2 |-> exp3]\n```	Supported; compiled as sorted slice of key-value structs

Function set (as function literal)	`[Nat -> Nat]`
	`\* or`
	`[Nat -> 1..3]`
		Unsupported

Function set (as struct)	`[1..3 -> 1..3]`	Unsupported

Function substitution	`[f EXCEPT ![exp1] = exp2]`
	`\* or`
	`[f EXCEPT !.field = exp]`
		Unsupported

If expression	`if condition`
	`    then thenExp`
	`    else elseExp`
		Compiled as
		
		`var result type;`
		`if condition {`
		`  result = thenExp`
		`} else {`
		`  result = elseExp`
		`}`
		`// result is used in place of`
		`// the expression hereafter`

Tuple (as slice)	`<<exp1, exp2, exp3>>`	Compiled as slice when all its contents are of the same type `[]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
Maybe action	`[exp1]_exp2`	Unsupported
Required action	`<<exp1>>_exp2`	Unsupported
Operator call	`op(arg1, arg2)`	Supported; compiled as expected

Quantified existential	`\E a \in exp1, b \in exp2 : exp3`	Compiled as
		
		`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`

Quantified universal	`\A a \in exp1, b \in exp2 : exp3`	Compiled as
		
		``
		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
		``

Set constructor	`{exp1, exp2, exp3}`	Compiled as sorted slice `[]type{exp1, exp2, exp3}`

Set comprehension	`{exp : a \in exp1, b \in exp2}`	Compiled as
		
		``
		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
		``

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

TLA+ constructs

3.2 Variable declarations

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

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

PlusCal

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

Compiled Go

3.3   Variable assignment

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

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

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

PlusCal

```
1  x, y = y, x+y
```

Compiled Go

3.4   Macros

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

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

PlusCal

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

Compiled Go

3.5   Data types

PGo supports PlusCal sets, functions, and tuples.

3.5.1   Sets

PlusCal sets are translated into sorted slices in Go.

9

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

PlusCal

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

Compiled Go

PlusCal also supports the typical mathematical set constructor notations.

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

PlusCal

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

<div align="center">Compiled Go</div>

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

3.5.2 Functions

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

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

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

<div align="center">PlusCal</div>

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

Compiled Go

3.5.3   Tuples

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

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

PlusCal

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

Compiled Go

3.6   Predicate operations

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

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

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

Compiled Go

3.7   With

The PlusCal `with` statement has the syntax

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

PlusCal

This construct selects the first element from each `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

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

Compiled Go

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

3.8 Processes

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

Uniprocess algorithms are translated into single-threaded Go programs.

In a multiprocess algorithm, processes can be declared with the syntax `process (Name \in S)` or `process (Name = Id)`. The first construct spawns a set of processes, one for each ID in the set `S`, and the second spawns a single process with ID `Id`. A process can refer to its identifier with the keyword `self`. In TLC, multiprocess algorithms are simulated by repeatedly selecting a random process then advancing it one atomic step (if it does not block).

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

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

PlusCal

3.8.1 Multi-threaded compilation strategy

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

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

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

Compiled multi-threaded Go program

3.8.2 Distributed process compilation strategy

TODO

3.9 Labels

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

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

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

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

<div align="center">PlusCal</div>

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

<div align="center">Compiled Go</div>

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

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   Example programs

4.1   Euclidean algorithm

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

```
1   ---------------------- MODULE Euclid --------------------------
2   EXTENDS Naturals, TLC
3   CONSTANT N
4
5   (*
6   --algorithm Euclid {
7     variables u = 24;
8             v \in 1 .. N;
9             v_init = v;
10    {
11    a: while (u # 0) {
12        if (u < v) {
13            u := v || v := u;
14        };
15    b: u := u - v;
16      };
17      print <<24, v_init, "have gcd", v>>
18    }
19  }
20  *)
21  \* BEGIN TRANSLATION
```

```
22  VARIABLES u, v, v_init, pc
23
24  vars == << u, v, v_init, pc >>
25
26  Init == (* Global variables *)
27         /\ u = 24
28         /\ v \in 1 .. N
29         /\ v_init = v
30         /\ pc = "a"
31
32  a == /\ pc = "a"
33      /\ IF u # 0
34            THEN /\ IF u < v
35                     THEN /\ /\ u' = v
36                             /\ v' = u
37                     ELSE /\ TRUE
38                             /\ UNCHANGED << u, v >>
39                  /\ pc' = "b"
40            ELSE /\ PrintT(<<24, v_init, "have gcd", v>>)
41                  /\ pc' = "Done"
42                  /\ UNCHANGED << u, v >>
43      /\ UNCHANGED v_init
44
45  b == /\ pc = "b"
46      /\ u' = u - v
47      /\ pc' = "a"
48      /\ UNCHANGED << v, v_init >>
49
50  Next == a \/ b
51           \/ (* Disjunct to prevent deadlock on termination *)
52              (pc = "Done" /\ UNCHANGED vars)
53
54  Spec == Init /\ [][Next]_vars
55
56  Termination == <>(pc = "Done")
57
58  \* END TRANSLATION
59  =====================================================================
```

PlusCal

```
1   package main
2
3   import (
4       "fmt"
5   )
6
7   var N int
8
9   func init() {
10      N = 8
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  }
```

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.

4.2 N-Queens problem

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

```
1  ------------------------ MODULE Queens ----------------------------
2  EXTENDS Naturals, Sequences, TLC
3  (***************************************************************************)
4  (* Formulation of the N-queens problem and an iterative algorithm to solve *)
5  (* the problem in TLA+. Since there must be exactly one queen in every row *)
6  (* we represent placements of queens as functions of the form *)
7  (* queens \in [ 1..N -> 1..N ] *)
8  (* where queens[i] gives the column of the queen in row i. Note that such *)
9  (* a function is just a sequence of length N. *)
10 (* We will also consider partial solutions, also represented as sequences *)
11 (* of length \leq N. *)
12 (***************************************************************************)
13
14 CONSTANT N \** number of queens and size of the board
15 ASSUME N \in Nat \ {0}
16
17 (* The following predicate determines if queens i and j attack each other
18    in a placement of queens (represented by a sequence as above). *)
19 Attacks(queens,i,j) ==
20   \/ queens[i] = queens[j] \** same column
21   \/ queens[i] - queens[j] = i - j \** first diagonal
22   \/ queens[j] - queens[i] = i - j \** second diagonal
23
24 (* A placement represents a (partial) solution if no two different queens
25    attack each other in it. *)
26 IsSolution(queens) ==
27   \A i \in 1 .. Len(queens)-1 : \A j \in i+1 .. Len(queens) :
28       ~ Attacks(queens,i,j)
29
30 (* Compute the set of solutions of the N-queens problem. *)
31 Solutions == { queens \in [1..N -> 1..N] : IsSolution(queens) }
32
33 (***************************************************************************)
34 (* We now describe an algorithm that iteratively computes the set of *)
```

```
35  (* solutions of the N-queens problem by successively placing queens. *)
36  (* The current state of the algorithm is given by two variables: *)
37  (* - todo contains a set of partial solutions, *)
38  (* - sols contains the set of full solutions found so far. *)
39  (* At every step, the algorithm picks some partial solution and computes *)
40  (* all possible extensions by the next queen. If N queens have been placed *)
41  (* these extensions are in fact full solutions, otherwise they are added *)
42  (* to the set todo. *)
43  (***************************************************************************)
44
45  (* --algorithm QueensPluscal
46      variables
47        todo = { << >> };
48        sols = {};
49
50      begin
51  nxtQ: while todo # {}
52        do
53          with queens \in todo,
54              nxtQ = Len(queens) + 1,
55              cols = { c \in 1..N : ~ \E i \in 1 .. Len(queens) :
56                              Attacks( Append(queens, c), i, nxtQ ) },
57              exts = { Append(queens,c) : c \in cols }
58          do
59            if (nxtQ = N)
60            then todo := todo \ {queens}; sols := sols \union exts;
61            else todo := (todo \ {queens}) \union exts;
62            end if;
63          end with;
64        end while;
65        print sols;
66      end algorithm
67  *)
68
69  \** BEGIN TRANSLATION
70  VARIABLES todo, sols, pc
71
72  vars == << todo, sols, pc >>
73
74  Init == (* Global variables *)
75          /\ todo = { << >> }
76          /\ sols = {}
77          /\ pc = "nxtQ"
78
79  nxtQ == /\ pc = "nxtQ"
80          /\ IF todo # {}
81                THEN /\ \E queens \in todo:
82                          LET nxtQ == Len(queens) + 1 IN
83                            LET cols == { c \in 1..N : ~ \E i \in 1 .. Len(queens) :
84                                              Attacks( Append(queens, c), i, nxtQ ) } IN
85                              LET exts == { Append(queens,c) : c \in cols } IN
86                                IF (nxtQ = N)
87                                    THEN /\ todo' = todo \ {queens}
88                                          /\ sols' = (sols \union exts)
89                                    ELSE /\ todo' = ((todo \ {queens}) \union exts)
90                                          /\ sols' = sols
91                      /\ pc' = "nxtQ"
92                ELSE /\ PrintT(sols)
93                      /\ pc' = "Done"
94                      /\ UNCHANGED << todo, sols >>
95
96  Next == nxtQ
```

```
97              \/ (* Disjunct to prevent deadlock on termination *)
98                 (pc = "Done" /\ UNCHANGED vars)
99
100  Spec == Init /\ [][Next]_vars
101
102  Termination == <>(pc = "Done")
103
104  \** END TRANSLATION
105
106  TypeInvariant ==
107    /\ todo \in SUBSET Seq(1 .. N) /\ \A s \in todo : Len(s) < N
108    /\ sols \in SUBSET Seq(1 .. N) /\ \A s \in sols : Len(s) = N
109
110  (* The set of sols contains only solutions, and contains all solutions
111     when todo is empty. *)
112  Invariant ==
113    /\ sols \subseteq Solutions
114    /\ todo = {} => Solutions \subseteq sols
115
116  (* Assert that no solutions are ever computed so that TLC displays one *)
117  NoSolutions == sols = {}
118
119  (* Add a fairness condition to ensure progress as long as todo is nonempty *)
120  Liveness == WF_vars(nxtQ)
121  LiveSpec == Spec /\ Liveness
122
123  =============================================================================
124  \* Modification History
125  \* Last modified Sat Jun 02 07:28:16 EDT 2018 by osboxes
126  \* Last modified Sat Dec 18 18:57:03 CET 2010 by merz
127  \* Created Sat Dec 11 08:50:24 CET 2010 by merz
```

PlusCal

```
1   package main
2
3   import (
4       "fmt"
5       "sort"
6   )
7
8   var N int
9
10  func init() {
11      N = 8
12  }
13
14  func Attacks(queens []int, i int, j int) bool {
15      return queens[i-1] == queens[j-1] || queens[i-1]-queens[j-1] == i-j || queens[j-1]-queens[i-1] == i-j
16  }
17
18  func main() {
19      todo := [][]int{[]int{}}
20      sols := [][]int{}
21      for {
22          if !(len(todo) != 0) {
23              break
24          }
25          queens := todo[0]
26          nxtQ := len(queens) + 1
27          tmpSet := make([]int, 0)
```

```go
28            tmpRange := make([]int, N-1+1)
29            for i := 1; i <= N; i++ {
30                tmpRange[i-1] = i
31            }
32            for _, c := range tmpRange {
33                exists := false
34                tmpRange0 := make([]int, len(queens)-1+1)
35                for i := 1; i <= len(queens); i++ {
36                    tmpRange0[i-1] = i
37                }
38                for _, i := range tmpRange0 {
39                    tmpSlice := make([]int, len(queens), len(queens)+1)
40                    copy(tmpSlice, queens)
41                    tmpSlice = append(tmpSlice, c)
42                    if Attacks(tmpSlice, i, nxtQ) {
43                        exists = true
44                        break
45                    }
46                }
47                if !exists {
48                    tmpSet = append(tmpSet, c)
49                }
50            }
51            cols := tmpSet
52            tmpSet0 := make([][]int, 0)
53            for _, c := range cols {
54                tmpSlice := make([]int, len(queens), len(queens)+1)
55                copy(tmpSlice, queens)
56                tmpSlice = append(tmpSlice, c)
57                tmpSet0 = append(tmpSet0, tmpSlice)
58            }
59            sort.Slice(tmpSet0, func(i int, j int) bool {
60                less := len(tmpSet0[i]) < len(tmpSet0[j])
61                if len(tmpSet0[i]) == len(tmpSet0[j]) {
62                    for i0 := 0; i0 < len(tmpSet0[i]); i0++ {
63                        less = tmpSet0[i][i0] < tmpSet0[j][i0]
64                        if tmpSet0[i][i0] != tmpSet0[j][i0] {
65                            break
66                        }
67                    }
68                }
69                return less
70            })
71            if len(tmpSet0) > 1 {
72                previousValue := tmpSet0[0]
73                currentIndex := 1
74                for _, v := range tmpSet0[1:] {
75                    eq := len(previousValue) == len(v)
76                    if eq {
77                        for i0 := 0; i0 < len(previousValue); i0++ {
78                            eq = previousValue[i0] == v[i0]
79                            if !eq {
80                                break
81                            }
82                        }
83                    }
84                    if !eq {
85                        tmpSet0[currentIndex] = v
86                        currentIndex++
87                    }
88                    previousValue = v
89                }
```

```go
 90                     tmpSet0 = tmpSet0[:currentIndex]
 91                 }
 92             exts := tmpSet0
 93             if nxtQ == N {
 94                 tmpSet1 := make([][]int, 0, len(todo))
 95                 for _, v := range todo {
 96                     eq := len(v) == len(queens)
 97                     if eq {
 98                         for i0 := 0; i0 < len(v); i0++ {
 99                             eq = v[i0] == queens[i0]
100                             if !eq {
101                                 break
102                             }
103                         }
104                     }
105                     if !eq {
106                         tmpSet1 = append(tmpSet1, v)
107                     }
108                 }
109                 todo = tmpSet1
110                 tmpSet2 := make([][]int, len(sols), len(sols)+len(exts))
111                 copy(tmpSet2, sols)
112                 tmpSet2 = append(tmpSet2, exts...)
113                 sort.Slice(tmpSet2, func(i0 int, j0 int) bool {
114                     less0 := len(tmpSet2[i0]) < len(tmpSet2[j0])
115                     if len(tmpSet2[i0]) == len(tmpSet2[j0]) {
116                         for i1 := 0; i1 < len(tmpSet2[i0]); i1++ {
117                             less0 = tmpSet2[i0][i1] < tmpSet2[j0][i1]
118                             if tmpSet2[i0][i1] != tmpSet2[j0][i1] {
119                                 break
120                             }
121                         }
122                     }
123                     return less0
124                 })
125                 if len(tmpSet2) > 1 {
126                     previousValue := tmpSet2[0]
127                     currentIndex := 1
128                     for _, v := range tmpSet2[1:] {
129                         eq := len(previousValue) == len(v)
130                         if eq {
131                             for i1 := 0; i1 < len(previousValue); i1++ {
132                                 eq = previousValue[i1] == v[i1]
133                                 if !eq {
134                                     break
135                                 }
136                             }
137                         }
138                         if !eq {
139                             tmpSet2[currentIndex] = v
140                             currentIndex++
141                         }
142                         previousValue = v
143                     }
144                     tmpSet2 = tmpSet2[:currentIndex]
145                 }
146                 sols = tmpSet2
147             } else {
148                 tmpSet1 := make([][]int, 0, len(todo))
149                 for _, v := range todo {
150                     eq := len(v) == len(queens)
151                     if eq {
```

23

```
152                     for i0 := 0; i0 < len(v); i0++ {
153                         eq = v[i0] == queens[i0]
154                         if !eq {
155                             break
156                         }
157                     }
158                 }
159                 if !eq {
160                     tmpSet1 = append(tmpSet1, v)
161                 }
162             }
163         tmpSet2 := make([][]int, len(tmpSet1), len(tmpSet1)+len(exts))
164         copy(tmpSet2, tmpSet1)
165         tmpSet2 = append(tmpSet2, exts...)
166         sort.Slice(tmpSet2, func(i0 int, j0 int) bool {
167             less0 := len(tmpSet2[i0]) < len(tmpSet2[j0])
168             if len(tmpSet2[i0]) == len(tmpSet2[j0]) {
169                 for i1 := 0; i1 < len(tmpSet2[i0]); i1++ {
170                     less0 = tmpSet2[i0][i1] < tmpSet2[j0][i1]
171                     if tmpSet2[i0][i1] != tmpSet2[j0][i1] {
172                         break
173                     }
174                 }
175             }
176             return less0
177         })
178         if len(tmpSet2) > 1 {
179             previousValue := tmpSet2[0]
180             currentIndex := 1
181             for _, v := range tmpSet2[1:] {
182                 eq := len(previousValue) == len(v)
183                 if eq {
184                     for i1 := 0; i1 < len(previousValue); i1++ {
185                         eq = previousValue[i1] == v[i1]
186                         if !eq {
187                             break
188                         }
189                     }
190                 }
191                 if !eq {
192                     tmpSet2[currentIndex] = v
193                     currentIndex++
194                 }
195                 previousValue = v
196             }
197             tmpSet2 = tmpSet2[:currentIndex]
198         }
199         todo = tmpSet2
200     }
201     }
202     fmt.Printf("%v\n", sols)
203 }
```

Compiled Go

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

4.3   Dijkstra's mutex algorithm

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

```
1 ------------------------- MODULE DijkstraMutex -------------------------
```

```
 2   (***************************************************************************)
 3   (* This is a PlusCal version of the first published mutual exclusion *)
 4   (* algorithm, which appeared in *)
 5   (* *)
 6   (* E. W. Dijkstra: "Solution of a Problem in Concurrent *)
 7   (* Programming Control". Communications of the ACM 8, 9 *)
 8   (* (September 1965) page 569. *)
 9   (* *)
10   (* Here is the description of the algorithm as it appeared in that paper. *)
11   (* The global variables are declared by *)
12   (* *)
13   (* Boolean array b, c[1:N]; integer k *)
14   (* *)
15   (* The initial values of b[i] and c[i] are true, for each i in 1..N. The *)
16   (* initial value of k can be any integer in 1..N. The pseudo-code for the *)
17   (* i-th process, for each i in 1..N, is: *)
18   (* *)
19   (* integer j; *)
20   (* Li0: b[i] := false; *)
21   (* Li1: if k # i then *)
22   (* Li2: begin c[i] := true; *)
23   (* Li3: if b[k] then k := i; *)
24   (* go to Li1 *)
25   (* end *)
26   (* else *)
27   (* Li4: begin c[i] := false; *)
28   (* for j := 1 step 1 until N do *)
29   (* if j # i and not c[j] then go to Li1 *)
30   (* end; *)
31   (* critical section; *)
32   (* c[i] := true; b[i] := true; *)
33   (* remainder of the cycle in which stopping is allowed; *)
34   (* go to Li0 *)
35   (* *)
36   (* It appears to me that the "else" preceding label Li4 begins the else *)
37   (* clause of the if statement beginning at Li1, and that the code from Li4 *)
38   (* through the end three lines later should be the body of that else *)
39   (* clause. However, the indentation indicates otherwise. Moreover, that *)
40   (* interpretation produces an incorrect algorithm. It seems that this *)
41   (* "else" actually marks an empty else clause for the if statement at Li1. *)
42   (* (Perhaps there should have been a semicolon after the "else".) *)
43   (***************************************************************************)
44
45   EXTENDS Integers
46
47   (***************************************************************************)
48   (* There is no reason why the processes need to be numbered from 1 to N. *)
49   (* So, we assume an arbitrary set Proc of process names. *)
50   (***************************************************************************)
51   CONSTANT Proc
52
53   (*********
54   Here is the PlusCal version of this algorithm.
55   The algorithm was modified from the original by adding a the variable temp2,
56    to avoid a type consistency conflict when temp changes type at Li4a.
57
58    --algorithm Mutex
59   { variables b = [i \in Proc |-> TRUE], c = [i \in Proc |-> TRUE], k \in Proc;
60     process (P \in Proc)
61        variable temp, temp2 ;
62        { Li0: while (TRUE)
63               { b[self] := FALSE;
```

```
64              Li1: if (k # self) { Li2: c[self] := TRUE;
65                              Li3a: temp := k;
66                              Li3b: if (b[temp]) { Li3c: k := self } ;
67                              Li3d: goto Li1
68                              };
69          Li4a: c[self] := FALSE;
70                  temp2 := Proc \ {self};
71          Li4b: while (temp2 # {})
72                  { with (j \in temp2)
73                      { temp2 := temp2 \ {j};
74                        if (~c[j]) { goto Li1 }
75                      }
76                  };
77           cs: skip; \* the critical section
78          Li5: c[self] := TRUE;
79          Li6: b[self] := TRUE;
80          ncs: skip \* non-critical section ("remainder of cycle")
81              }
82      }
83  }
84  Notes on the PlusCal version:
85
86  1. Label Li3d is required by the translation. It could be eliminated by
87     adding a then clause to the if statement of Li3b and putting the goto
88     in both branches of the if statement.
89
90  2. The for loop in section Li4 of the original has been changed to
91     a while loop that examines the other processes in an arbitrary
92     (nondeterministically chosen) order. Because temp is set equal
93     to the set of all processes other than self, there is no need for
94     a test corresponding to the "if j # i" in the original. Note that
95     the process-local variable j has been replaced by the identifier
96     j that is local to the with statement.
97  *********)
98  \* BEGIN TRANSLATION
99  CONSTANT defaultInitValue
100 VARIABLES b, c, k, pc, temp
101
102 vars == << b, c, k, pc, temp >>
103
104 ProcSet == (Proc)
105
106 Init == (* Global variables *)
107         /\ b = [i \in Proc |-> TRUE]
108         /\ c = [i \in Proc |-> TRUE]
109         /\ k \in Proc
110         (* Process P *)
111         /\ temp = [self \in Proc |-> defaultInitValue]
112         /\ pc = [self \in ProcSet |-> CASE self \in Proc -> "Li0"]
113
114 Li0(self) == /\ pc[self] = "Li0"
115             /\ b' = [b EXCEPT ![self] = FALSE]
116             /\ pc' = [pc EXCEPT ![self] = "Li1"]
117             /\ UNCHANGED << c, k, temp >>
118
119 Li1(self) == /\ pc[self] = "Li1"
120             /\ IF k # self
121                   THEN /\ pc' = [pc EXCEPT ![self] = "Li2"]
122                   ELSE /\ pc' = [pc EXCEPT ![self] = "Li4a"]
123             /\ UNCHANGED << b, c, k, temp >>
124
125 Li2(self) == /\ pc[self] = "Li2"
```

26

```
126               /\ c' = [c EXCEPT ![self] = TRUE]
127               /\ pc' = [pc EXCEPT ![self] = "Li3a"]
128               /\ UNCHANGED << b, k, temp >>
129
130 Li3a(self) == /\ pc[self] = "Li3a"
131               /\ temp' = [temp EXCEPT ![self] = k]
132               /\ pc' = [pc EXCEPT ![self] = "Li3b"]
133               /\ UNCHANGED << b, c, k >>
134
135 Li3b(self) == /\ pc[self] = "Li3b"
136               /\ IF b[temp[self]]
137                     THEN /\ pc' = [pc EXCEPT ![self] = "Li3c"]
138                     ELSE /\ pc' = [pc EXCEPT ![self] = "Li3d"]
139               /\ UNCHANGED << b, c, k, temp >>
140
141 Li3c(self) == /\ pc[self] = "Li3c"
142               /\ k' = self
143               /\ pc' = [pc EXCEPT ![self] = "Li3d"]
144               /\ UNCHANGED << b, c, temp >>
145
146 Li3d(self) == /\ pc[self] = "Li3d"
147               /\ pc' = [pc EXCEPT ![self] = "Li1"]
148               /\ UNCHANGED << b, c, k, temp >>
149
150 Li4a(self) == /\ pc[self] = "Li4a"
151               /\ c' = [c EXCEPT ![self] = FALSE]
152               /\ temp' = [temp EXCEPT ![self] = Proc \ {self}]
153               /\ pc' = [pc EXCEPT ![self] = "Li4b"]
154               /\ UNCHANGED << b, k >>
155
156 Li4b(self) == /\ pc[self] = "Li4b"
157               /\ IF temp[self] # {}
158                     THEN /\ \E j \in temp[self]:
159                                 /\ temp' = [temp EXCEPT
160                                               ![self] = temp[self] \ {j}]
161                                 /\ IF ~c[j]
162                                       THEN /\ pc' = [pc EXCEPT
163                                                       ![self] = "Li1"]
164                                       ELSE /\ pc' = [pc EXCEPT
165                                                       ![self] = "Li4b"]
166                     ELSE /\ pc' = [pc EXCEPT ![self] = "cs"]
167                             /\ UNCHANGED temp
168               /\ UNCHANGED << b, c, k >>
169
170 cs(self) == /\ pc[self] = "cs"
171             /\ TRUE
172             /\ pc' = [pc EXCEPT ![self] = "Li5"]
173             /\ UNCHANGED << b, c, k, temp >>
174
175 Li5(self) == /\ pc[self] = "Li5"
176             /\ c' = [c EXCEPT ![self] = TRUE]
177             /\ pc' = [pc EXCEPT ![self] = "Li6"]
178             /\ UNCHANGED << b, k, temp >>
179
180 Li6(self) == /\ pc[self] = "Li6"
181             /\ b' = [b EXCEPT ![self] = TRUE]
182             /\ pc' = [pc EXCEPT ![self] = "ncs"]
183             /\ UNCHANGED << c, k, temp >>
184
185 ncs(self) == /\ pc[self] = "ncs"
186             /\ TRUE
187             /\ pc' = [pc EXCEPT ![self] = "Li0"]
```

```
188 |             /\ UNCHANGED << b, c, k, temp >>
189 |
190 | P(self) == Li0(self) \/ Li1(self) \/ Li2(self) \/ Li3a(self) \/ Li3b(self)
191 |               \/ Li3c(self) \/ Li3d(self) \/ Li4a(self) \/ Li4b(self)
192 |               \/ cs(self) \/ Li5(self) \/ Li6(self) \/ ncs(self)
193 |
194 | Next == (\E self \in Proc: P(self))
195 |            \/ (* Disjunct to prevent deadlock on termination *)
196 |               ((\A self \in ProcSet: pc[self] = "Done") /\ UNCHANGED vars)
197 |
198 | Spec == Init /\ [][Next]_vars /\ \A self \in Proc: WF_vars(P(self))
199 |
200 | Termination == <>(\A self \in ProcSet: pc[self] = "Done")
201 |
202 | \* END TRANSLATION
203 |
204 | (***************************************************************************)
205 | (* The following formula asserts that no two processes are in their *)
206 | (* critcal sections at the same time. It is the invariant that a mutual *)
207 | (* exclusion algorithm should satisfy. You can have TLC check that the *)
208 | (* algorithm is a mutual exclusion algorithm by checking that this formula *)
209 | (* is an invariant. *)
210 | (***************************************************************************)
211 | MutualExclusion == \A i, j \in Proc :
212 |                    (i # j) => ~ /\ pc[i] = "cs"
213 |                                 /\ pc[j] = "cs"
214 | (***************************************************************************)
215 | (* An equivalent way to perform the same test would be to change the *)
216 | (* statement labeled cs (the critical section) to *)
217 | (* *)
218 | (* cs: assert \A j \in Proc \ {self} : pc[j] # "cs" *)
219 | (* *)
220 | (* You can give this a try. However, the assert statement requires that *)
221 | (* the EXTENDS statement also import the standard module TLC, so it should *)
222 | (* read *)
223 | (* *)
224 | (* EXTENDS Integers, TLC *)
225 | (***************************************************************************)
226 |
227 | -------------------------------------------------------------------------
228 |
229 | (***************************************************************************)
230 | (* LIVENESS *)
231 | (* *)
232 | (* If you are a sophisticated PlusCal user and know a little temporal *)
233 | (* logic, you can continue reading about the liveness properties of the *)
234 | (* algorithm. *)
235 | (* *)
236 | (* Dijkstra's algorithm is "deadlock free", which for a mutual exclusion *)
237 | (* algorithm means that if some process is trying to enter its critical *)
238 | (* section, then some process (not necessarily the same one) will *)
239 | (* eventually enter its critical section. Since a process begins trying *)
240 | (* to enter its critical section when it is at the control point labeled *)
241 | (* Li0, and it is in its critical section when it is at control point cs, *)
242 | (* the following formula asserts deadlock freedom. *)
243 | (***************************************************************************)
244 | DeadlockFree == \A i \in Proc :
245 |                   (pc[i] = "Li0") ~> (\E j \in Proc : pc[j] = "cs")
246 | (***************************************************************************)
247 | (* Dijkstra's algorithm is deadlock free only under the assumption of *)
248 | (* fairness of process execution. The simplest such fairness assumption *)
249 | (* is weak fairness on each process's next-state action. This means that *)
```

```
250  (* no process can halt if it is always possible for that process to take a *)
251  (* step. The following statement tells the PlusCal translator to define *)
252  (* the specification to assert weak fairness of each process's next-state *)
253  (* action. *)
254  (* *)
255  (* PlusCal options (wf) *)
256  (* *)
257  (* This statement can occur anywhere in the file--either in a comment or *)
258  (* before or after the module. Because of this statement, the translator *)
259  (* has added the necessary fairness conjunct to the definition of Spec. *)
260  (* So, you can have the TLC model checker check that the algorithm *)
261  (* satisfies property DeadlockFree. *)
262  (***************************************************************************)
263
264  (***************************************************************************)
265  (* Dijkstra's algorithm is not "starvation free", because it allows some *)
266  (* waiting processes to "starve", never entering their critical section *)
267  (* while other processes keep entering and leaving their critical *)
268  (* sections. Starvation freedom is asserted by the following formula. *)
269  (* You can use TLC to show that the algorithm is not starvation free by *)
270  (* producing a counterexample trace. *)
271  (***************************************************************************)
272  StarvationFree == \A i \in Proc :
273                       (pc[i] = "Li0") ~> (pc[i] = "cs")
274
275  (***************************************************************************)
276  (* In this algorithm, no process can ever be blocked waiting at an 'await' *)
277  (* statement or a 'with (v \in S)' statement with S empty. Therefore, *)
278  (* weak fairness of each process means that each process keeps continually *)
279  (* trying to enter its critical section, and it exits the critical *)
280  (* section. An important requirement of a mutual exclusion solution, one *)
281  (* that rules out many simple solutions, is that a process is allowed to *)
282  (* remain forever in its non-critical section. (There is also no need to *)
283  (* require that a process that enters its critical section ever leaves it, *)
284  (* though without that requirement the definition of starvation freedom *)
285  (* must be changed.) *)
286  (* *)
287  (* We can allow a process to remain forever in its critical section by *)
288  (* replacing the 'skip' statement that represents the non-critical section *)
289  (* with the following statement, which allows the process to loop forever. *)
290  (* *)
291  (* ncs: either skip or goto ncs *)
292  (* *)
293  (* An equivalent non-critical section is *)
294  (* *)
295  (* nsc: either skip or await FALSE *)
296  (* *)
297  (* A more elegant method is to change the fairness requirement to assert *)
298  (* weak fairness of a process's next-state action only when the process is *)
299  (* not in its non-critical section. This is accomplished by taking the *)
300  (* following formula LSpec as the algorithm's specification. *)
301  (***************************************************************************)
302  LSpec == Init /\ [][Next]_vars
303             /\ \A self \in Proc: WF_vars((pc[self] # "ncs") /\ P(self))
304
305  (***************************************************************************)
306  (* If we allow a process to remain forever in its non-critical section, *)
307  (* then our definition of deadlock freedom is too weak. Suppose process p *)
308  (* were in its critical section and process q, trying to enter its *)
309  (* critical section, reached Li1. Formula DeadlockFree would allow a *)
310  (* behavior in which process q exited its critical section and remained *)
311  (* forever in its non-critical section, but process p looped forever *)
```

```
312  (* trying to enter its critical section and never succeeding. To rule out *)
313  (* this possibility, we must replace the formula *)
314  (* *)
315  (* pc[i] = "Li0" *)
316  (* *)
317  (* in DeadLock free with one asserting that control in process i is *)
318  (* anywhere in control points Li0 through Li4b. It's easier to express *)
319  (* this by saying where control in process i is NOT, which we do in the *)
320  (* following property. *)
321  (****************************************************************************)
322  DeadlockFreedom ==
323      \A i \in Proc :
324        (pc[i] \notin {"Li5", "Li6", "ncs"}) ~> (\E j \in Proc : pc[j] = "cs")
325  (****************************************************************************)
326  (* Do you see why it's not necessary to include "cs" in the set of values *)
327  (* that pc[i] does not equal? *)
328  (****************************************************************************)
329
330
331
332  (****************************************************************************)
333  (* Using a single worker thread on a 2.5GHz dual-processor computer, TLC *)
334  (* can check MutualExclusion and liveness of a 3-process model in about 2 *)
335  (* or 3 minutes (depending on which spec is used and which liveness *)
336  (* property is checked). That model has 90882 reachable states and a *)
337  (* state graph of diameter 54. TLC can check a 4-process model in about *)
338  (* 53 minutes. That model has 33288512 reachable states and a state graph *)
339  (* of diameter 89. *)
340  (****************************************************************************)
341  ============================================================================
342  \* Modification History
343  \* Last modified Sun Dec 31 22:04:29 EST 2017 by osboxes
344  \* Last modified Sat Jan 01 12:14:14 PST 2011 by lamport
345  \* Created Fri Dec 31 14:14:14 PST 2010 by lamport
```

PlusCal

```go
 1  package main
 2
 3  import (
 4      "sort"
 5      "sync"
 6  )
 7
 8  var Proc []int
 9
10  var b []struct {
11      key   int
12      value bool
13  }
14
15  var c []struct {
16      key   int
17      value bool
18  }
19
20  var k int
21
22  var pGoStart chan bool
23
24  var pGoWait sync.WaitGroup
```

```go
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[0].Lock()
    for {
        if !true {
            pGoLock[0].Unlock()
            break
        }
        key := self
        index := sort.Search(len(b), func(i int) bool {
            return !(b[i].key < key)
        })
        b[index].value = false
        pGoLock[0].Unlock()
    Li1:
        pGoLock[2].Lock()
        if k != self {
            pGoLock[2].Unlock()
            pGoLock[1].Lock()
            key0 := self
            index0 := sort.Search(len(c), func(i0 int) bool {
                return !(c[i0].key < key0)
            })
```

```go
        c[index0].value = true
        pGoLock[1].Unlock()
        pGoLock[2].Lock()
        temp = k
        pGoLock[2].Unlock()
        pGoLock[0].Lock()
        key1 := temp
        index1 := sort.Search(len(b), func(i1 int) bool {
            return !(b[i1].key < key1)
        })
        if b[index1].value {
            pGoLock[0].Unlock()
            pGoLock[2].Lock()
            k = self
            pGoLock[2].Unlock()
        } else {
            pGoLock[0].Unlock()
        }
        goto Li1
    } else {
        pGoLock[2].Unlock()
    }
    pGoLock[1].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[1].Unlock()
    pGoLock[1].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[1].Unlock()
            goto Li1
        }
    }
    pGoLock[1].Unlock()
    pGoLock[1].Lock()
    key1 := self
    index1 := sort.Search(len(c), func(i1 int) bool {
```

```
149            return !(c[i1].key < key1)
150        })
151        c[index1].value = true
152        pGoLock[1].Unlock()
153        pGoLock[0].Lock()
154        key2 := self
155        index2 := sort.Search(len(b), func(i2 int) bool {
156            return !(b[i2].key < key2)
157        })
158        b[index2].value = true
159        pGoLock[0].Unlock()
160        pGoLock[0].Lock()
161    }
162    pGoLock[0].Unlock()
163 }
164
165 func main() {
166    for _, v := range Proc {
167        pGoWait.Add(1)
168        go P(v)
169    }
170    close(pGoStart)
171    pGoWait.Wait()
172 }
```

Compiled Go

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