PGo Manual

For commit 90094d5b

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.

```
{
    "build": {
        "output_dir": "",
        "dest_file": ""
    },
    "networking": {
        "enabled": true,
        "state": {
            "strategy": "",
            "endpoints": [],
            "peers": [],
            "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 `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.

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

Example constant specification

```go
var ProcSet []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 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

<table>
<tr><td>While loop</td><td><pre><code>while (condition) {
  body;
}</code></pre></td><td>Compiled as<pre><code>for {
  if !condition {
    break
  }
  body
}</code></pre></td></tr>
<tr><td>If statement</td><td><pre><code>if (condition) {
  thenPart;
} else {
  elsePart;
}</code></pre></td><td>Supported; compiled as expected</td></tr>
<tr><td>Either statement</td><td><pre><code>either { stmt1; stmt2; }
    or { stmt3; stmt4; }
    or { stmt5; stmt6; }
    * ...</code></pre></td><td>Compiled as<pre><code>case0:
  stmt1
  stmt2
case1:
  stmt3
  stmt4
case2:
  stmt5
  stmt6
// ...
endEither:</code></pre></td></tr>
<tr><td>Assignment</td><td><code>x := exp;</code></td><td>Supported; compiled as expected</td></tr>
<tr><td>Multiple variable assignment</td><td><code>x := y || y := x + y;</code></td><td>Supported; compiled as multiple assignment in Go</td></tr>
<tr><td>Return statement</td><td><code>return;</code></td><td>Supported; compiled as expected</td></tr>
<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><code>with (x = exp1, y \in exp2) {
  body;
}</code></pre></td><td>Supported; compiled as variable assignment with fresh names</td></tr>
<tr><td>Print statement</td><td><code>print exp;</code></td><td>Supported; compiled as expected</td></tr>
<tr><td>Assert statement</td><td><code>assert condition;</code></td><td>Compiled as<pre><code>if !condition {
  panic("condition");
}</code></pre></td></tr>
</table>

<table>
<tr><td>Await statement</td><td><pre>await condition;</pre></td><td>Compiled as<pre>awaitLabel:
  if !condition {
    goto awaitLabel
  }</pre></td></tr>
<tr><td>Goto statement</td><td><pre>goto label;</pre></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

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

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]`.

```
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

```go
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

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.

```
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 $\forall$ and $\exists$. 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`. 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:

```
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

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.

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

import (
    "fmt"
)

var N int

func init() {
    N = 8
}

func main() {
    u := 24
    tmpRange := make([]int, N-1+1)
    for i := 1; i <= N; i++ {
        tmpRange[i-1] = i
    }
    v := tmpRange[0]
    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.

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.

```
-------------------------- 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)
```

```go
		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++
                    }
                    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.

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.

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

```go
package main

import (
    "sort"
    "sync"
)

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
```

```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)
			})
```

```
            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 {
```

```
            return !(c[i1].key < key1)
        })
        c[index1].value = true
        pGoLock[1].Unlock()
        pGoLock[0].Lock()
        key2 := self
        index2 := sort.Search(len(b), func(i2 int) bool {
            return !(b[i2].key < key2)
        })
        b[index2].value = true
        pGoLock[0].Unlock()
        pGoLock[0].Lock()
    }
    pGoLock[0].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.