vgo Documentation

Release 0.0.1

Justus Winter

September 25, 2016

Contents

Contents:

CHAPTER 1

vgo - verifiable go

Verifiable go or vgo for short is a subset of the go programming language introduced by Google that has been extended so that the programmer can use a temporal logic (CTL) to express assumptions and guarantees directly within the source code.

The tool is invoked like the go build system itself. It verifies that the code has the desired properties and produces a compiled library or aborts with an appropriate error message.

This has a number of advantages:

- The tool fits very well into the usual development practices.
- Verification failures prevent compilation of the code.
- The very same code that has been model checked is compiled, there is no need to translate the code to an intermediate language.

The produced library is a standard go module that other code written in go can use. This makes it possible to divide the code into parts that can be verified (like the core logic) and parts that is not easily verifiable (any kind of I/O, user interfaces, etc.).

1.1 Goals

The goals of vgo are

- to provide an implementation of a modular model checking framework for further experimentation and
- to provide a tool capable of checking properties of programs written in a modern imperative programming language while focusing on usability aspects, i.e. to provide a tool that can be used by programmers without intimate knowledge of the theoretical background of model checking.

CHAPTER 2

Installing vgo

2.1 Obtaining vgo

The source code of vgo is available via mercurial. Provided that you have mercurial installed, you can obtain vgo by doing:

```
% hg clone https://bitbucket.org/teythoon/vgo
destination directory: vgo
requesting all changes
adding changesets
adding manifests
adding file changes
[...]
```

2.2 Compiling vgo

In order to compile vgo you need the golang tool chain. On Debian systems you can install it using *aptitude install golang*.

Go to your vgo checkout and do:

```
% make
GOPATH=[...] go install vgo
[...]
```

2.3 Running the test suite

To make sure that your build of vgo works as expected, you can optionally run the test suite:

```
% make test
[...]
170 / 170 tests successful.
```

2.4 Installing vgo

You can optionally install vgo to a location of your choice by running:

```
% make install PREFIX=/usr/local/stow/vgo
[...]
```

2.5 Using vgo

All of vgos functionality can be invoked using the *vgo* binary. Running it without arguments to get a list of available commands:

```
vgo is a frontend for the verifiable go toolchain.

Usage:

        vgo [vgo options] command [command options] [command args]

The commands are:

    install     verifies and installs the given vgo package
    inspect     vgo reports interactively
    runtests    runs the vgo test suite
    config      prints the vgo configuration
    version     prints the vgo version
    help        prints usage and flags of a given command

Use "vgo help [command]" for more information about a command.

Additional help topics:

    options     global vgo options

Use "vgo help [topic]" for more information about that topic.
```

vgo needs its source code to translate model checking stubs in the verification step. It uses the environment variable *VGOROOT* to locate it. If *VGOROOT* is not set, it defaults to */usr/lib/vgo*. As a convenience if you run vgo directly from the source tree it uses this location automatically.

If you have installed vgo to *${PREFIX}*, you must set *VGOROOT* to *${PREFIX}/lib/vgo*. Make sure that the vgo frontend *vgo* is in your *${PATH}*. With bourne shell like shells this can be done using:

```
% export VGOROOT="${PREFIX}/lib/vgo"
% export PATH="${PREFIX}/bin:${PATH}"
% vgo runtests
[...]
170 / 170 tests successful.
```

CHAPTER 3

The *vgo* tutorial

Welcome to the *vgo* tutorial. You will need a working *vgo* installation and either a *vgo* source checkout or the *vgo-doc* package installed. To generate graphs the *graphviz* package is needed. On Debian like systems you can install it using *aptitude install graphviz*.

You will find the source code for each example here either in the source tree under *docs/tutorial* or */usr/share/doc/vgo-doc/tutorial*.

3.1 Getting started

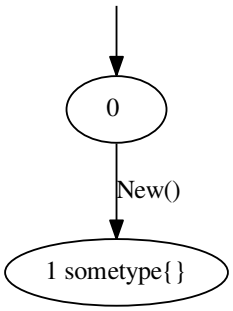
The source for this example is in the directory *tutorial0*. Copy this directory and fire up *vgo*:

```
% cp --recursive /usr/share/doc/vgo-doc/tutorial/tutorial0 /tmp
% cd /tmp/tutorial0
% vgo install -report gettingstarted
02:35:31 [Info] Welcome to the one and only model checker!
02:35:31 [Info] vgo version 0.0.1.44202be27a4b+
02:35:31 [Info] Writing report into /tmp/vgo-install126016440
02:35:32 [Info] Constructing model for type sometype
02:35:32 [Info] sometype: Exploring state space
02:35:32 [Info] Found 2 distinct states connected by 1 edges
02:35:32 [Info] sometype: State space explored
02:35:32 [Info] Package gettingstarted installed successfully
```

vgo explores the graph of reachable states, also known as *Kripke structure*, and renders it as a scalable vector graphic if asked to do so using the *-report* flag. Let us have a look at this image:

```
% xdg-open /tmp/vgo-install126016440/sometype.svg
```

The image should look like this:



The each node in this diagram is a state of the program. The state 0 is special though, it is there to be a common parent to all objects created by the constructors. There is only one constructor and it allocates *sometype* objects using *new*. Since the memory is zeroed, the field *a* is set to *false* (the zeroish value of type *bool*).

So this is not a very interesting program, it has just one state since there is no way to modify the non-exported field *a*. But it is a start. Let us look at it in more detail.

Each node is a reachable state and has a unique number. It is also labeled with the string representation of the object representing that state. This string representation uses an optimization for boolean fields, if a field is *true*, its name is included, if it is *false*, it is omitted. Since the string representation is *sometype{}*, the field *a* must be *false* in this state.

Every edge indicates a possible state transition and is annotated with the method that invokes this state transition. In this example the only edge is the one of the *New()* constructor.

Let us look at the source. Open *src/gettingstarted/main.vgo* with your favorite editor and enable the *go* syntax highlighting mode. You will see this:

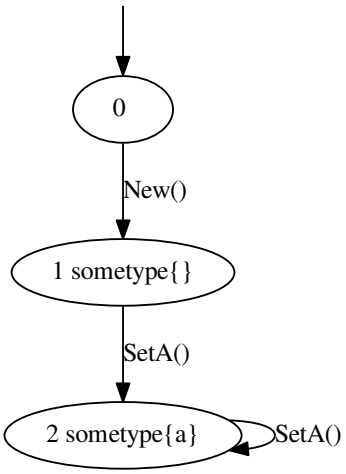
```
package gettingstarted

type sometype struct {
    a bool
}

func New() *sometype {
    return new(sometype)
}
```

This is both a valid *vgo* and a valid *go* program. *vgo* is a superset of a subset of *go*. It starts with a package declaration and defines a type *sometype* with a private member *a* of type *bool*.

Let us now add a method to make the program more interesting. Add the following to the file *main.vgo*:

```
func (s *sometype) SetA() {
    s.a = true
}
```

If you recompile the program using *vgo*, you will see:



So we added a method *SetA()* that sets the member *a* to true. Consequently we now see a second state 2 that can be reached from the initially constructed state 1 by invoking the *SetA* method. If we are in state 2 and invoke this method again, nothing changes.

Now let us annotate our type with guarantees that can be verified using the model checking engine in *vgo*. Let us specify, that on all (infinite) paths there is some point in the future where the member *a* is *true*. The corresponding CTL formula is AFa. Extend the *vgo* type declaration to read:

```
type sometype struct {
    a bool
} satisfying {
    AF`s.a`
}
```

In *vgo* atomic propositions are enclosed in backticks (‘) and must be *go* expressions of type boolean. The current object is bound to *s*, so *s.a* accesses the member *a* of type boolean (in case you are not familiar with *go* note that it is customary to name this reference like the first letter of your type converted to lower case, so it is a *s* because the type is called *sometype*). If you recompile the program you will see:

```
% vgo install -report gettingstarted
03:30:08 [Info] Welcome to the one and only model checker!
03:30:08 [Info] vgo version 0.0.1.44202be27a4b+
03:30:08 [Info] Writing report into /tmp/vgo-install608100277
03:30:09 [Info] Constructing model for type sometype
03:30:09 [Info] sometype: Exploring state space
03:30:09 [Info] Found 3 distinct states connected by 3 edges
03:30:09 [Info] sometype: State space explored
03:30:09 [Info] conventional.Check(sometype, AF`s.a`) successful.
03:30:09 [Info] Package gettingstarted installed successfully
```

So the program was found to be fulfilling the specification. Go ahead and add another guarantee like $EFAGa$.

3.2 The microwave oven

Microwave ovens are a canonical model checking showcase, so let us implement one. The source for this example is in *tutorial1*, you can compile it using *vgo install microwave*. If you get stuck in this section, you can look at the microwave example shipped with vgo under *src/vgo/tests/microwave*.

If you look at the source you will find the following type declaration:

```
type microwave struct {
    on bool      // is the microwave on?
    open bool    // is the door open?
    error string // was the microwave used incorrectly? how?
} satisfying {
    AG!(`m.DoorOpen()` & `m.On()`) // the microwave is never on if the
                                   // door is open

    AG!(`m.Error()` & `m.On()`)    // the microwave is never on if the
                                   // error flag is set

    AGEF!(`m.Error()` | (`m.On()` | `m.DoorOpen()`)) // resettable

    AGAF!`m.On()` // the microwave is never on forever
}
```

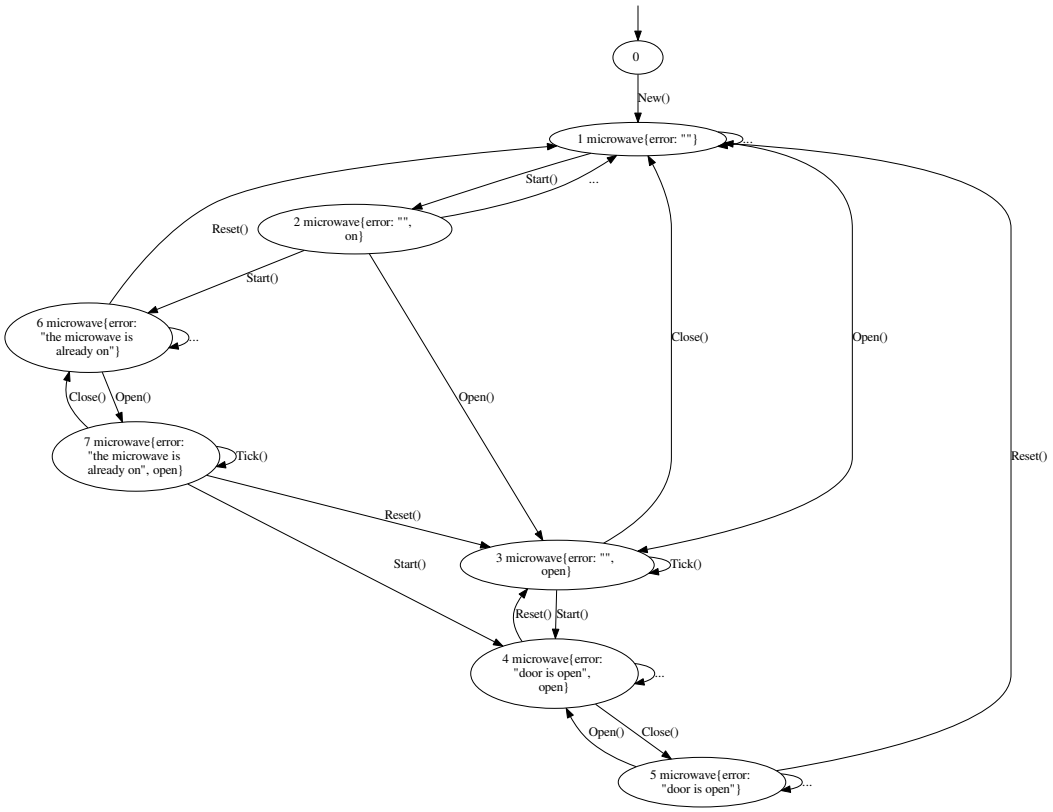
Currently the microwave fulfills the specification, but it is a not a very useful microwave since one cannot turn it on. Please implement the methods *Start()*, *Tick()*, *Open()*, *Close()*, *Reset()* and *setError(s)*. Some remarks:

- It should be considered an error if one invokes *Start()* while the door is open.
- *Tick()* models elapsing time. Cooking should require one unit of time. The bool field *on* models this just fine.
- There are two kinds of errors here:
- Imagine a microwave with the door closed. Now a user can open the door and she gets a microwave with the door being open. It is not possible to open the door again, since it is already open. The *Open()* function should reflect this by returning an *error*. Note that by returning an *error* you indicate to *vgo* that you do not change state of the object.
- But if on the other hand she presses the start button while the door is still open, we want to inform her of this error by displaying it and make her acknowledge the error by resetting the microwave. The *Start()* function should set the *error* field to an appropriate message using *setError(s)*.

The *vgo* frontend has a switch to display verification reports using your browser:

```
% vgo install -inspect microwave
```

For reference, here is the *Kripke structure* of the microwave in *src/vgo/examples/microwave* which fulfills the specification:



3.3 Numerical values and constructors

So far we only used types that were structs containing booleans. But one of the cool features of *go* (and thus *vgo*) is the fact that you can define specialized versions of primitive data types and bind methods to them.

Model Checking suffers from the so called *state explosion problem*. This problem gets worse if we use numeric data types because there are 2^{32} possible 32 bit integers and thus possible states instead of the two possible values of a boolean. Exploring the state space of a program using a single 32 bit integer is not feasible if done naively, but there are ways to work around that (e.g. *abstract interpretation*). Currently *vgo* handles only small integers, but we can do some interesting things nonetheless.

Let us have a look at *tutorial2/src/collatz/collatz.vgo*:

```
package collatz

import "fmt"

type collatz uint8 satisfying {
    AGAF`*c == 1`
}

// New constructs objects of type collatz.
func New(n uint8) (*collatz, error) {
    if n == 0 || n > 9 {
        return nil, fmt.Errorf("too large: %v", n)
```

```
    }

    c := collatz(n)
    return &c, nil
}

// Step computes one step of the collatz conjecture, i.e. it changes c
// to c/2 if c is even else c*3+1.
func (c *collatz) Step() {
    // implement this function
}
```

Some notes:

- In the declaration of *Step()*, *c* is called the receiver.
- As before the receiver is a pointer type (*c *Collatz*), so you can use the dereference operator * to access and modify its value (e.g. **c = 5*).
- If you haven't heard of the *Collatz conjecture*, you can consult Wikipedia.
- You can use the *-compact-graphs* flag to make *vgo* create more compact graphs by omitting the numerical node ids and edge labels.

Tasks:

- Implement *Step()*. Check whether your work fulfills the specification.

3.4 The *Nim* game

So far the examples have been somewhat constructed and we haven't seen the code run. So let us consider a more interesting example and implement the *Nim* game. The code for this section is in *tutorial/tutorial3*, it contains a *Nim* engine written in *vgo* (*src/engine*) and a console application written in *go* using the engine (*src/nim*). If you get stuck in this section, you can look at *src/vgo/tests/engine*.

This example highlights an important concept of *vgo*, namely the separation of programs into parts that can be model checked (i.e. the game engine) and parts that cannot easily be model checked (i.e. the frontend).

The most interesting part is in *src/engine/engine.vgo*. The type declaration reads:

```
type nim struct {
    heaps [3]uint8 // the three heaps
    turn Player    // turn indicates whos turn it is
} satisfying {
    !EF`n.Evaluate() == OPPONENT` // the engines opponent never wins

    AGEF`n.Evaluate() == ENGINE`  // there is always a path where
                                  // eventually the engine has won
}
```

Some notes:

- *Move(*Move) error* is used to execute moves for the engines opponent. If the move is not valid in the current game state, it returns an error.
- *Ponder() (*Move, error)* uses *ponder() (*Move, error)* to generate a move for the engine and executes and returns the move.
- The type *nim* has two guarantees stating that there is no reachable state in which the opponent (i.e. you) has won and that in every state there is always a path where in the future the engine has won.

- *ponder()* currently generates an arbitrary valid move.
- There is a Makefile, you can execute *make* to build the engine and (provided the verification was successful) the frontend *bin/nim*.
- In *go* and thus *vgo* the operator for exclusive or is ^.

Tasks:

- Implement the winning strategy described below in *ponder()*.
- (optional) Create a modified version for the misère version of the game.
- (optional) Modify the game to handle arbitrary starting positions.

Some theory:

- The key to winning the game is to finish every move with a *nim sum* over all stacks of 0. The *nim sum* is binary exclusive or.
- If the *nim sum* of all the stacks is not 0, it is always possible to make a move to make the sum 0. In the default starting position (3, 4, 5) the *nim sum* is not 0, so the first player has a winning strategy.

The winning strategy is:

- Compute *s*, the *nim sum* of all heaps.
- Iterate over the heaps
  - test whether *count* ^ *s* < *count*. If this is the case, return the move that removes *count* - *(count* ^ *s)* objects from that heap. This results in a state where the *nim sum* is 0.

CHAPTER 4

Indices and tables

- genindex
- modindex
- search